System Resilience Amplify Failures, Detect, or Both?

(A ROSS'19 Invited Talk)

Arnab Das, Ian Briggs, Mark Baranowski, Vishal Sharma
Zvonimir Rakamaric, Sriram Krishnamoorthy, Ganesh Gopalakrishnan
University of Utah, School of Computing (plus PNNL, Microsoft)
http://www.cs.utah.edu/~ganesh
http://www.parallel.utah.edu

System Resilience: Need

TOWARD EXASCALE RESILIENCE

Franck Cappello[1]
Al Geist[2]
Bill Gropp[3]
Laxmikant Kale[3]
Bill Kramer[4]
Marc Snir[3]

Abstract

Over the past few years resilience has became a major issue for high-performance computing (HPC) systems, in particular in the perspective of large petascale systems and future exascale systems. These systems will typically gather from half a million to several millions of central processing unit (CPU) cores running up to a billion threads. From the current knowledge and observations of existing large systems, it is anticipated that exascale systems will experience various kind of faults many times per day. It is also anticipated that the current approach for resilience, which relies on automatic or application level checkpoint/ restart, will not work because the time for checkpointing and restarting will exceed the mean time to failure of a full system. This set of projections leaves the community of

Soft-error resilience of the IBM POWER6 processor

P. N. Sanda
J. W. Kellington
P. Kudva
R. Kalla
R. B. McBeth
J. Ackaret
R. Lockwood
J. Schumann
C. R. Jones

The error detection and correction capability of the IBM POWER6™ processor enables high tolerance to single-event upsets. The soft-error resilience was tested with proton beam- and neutron beam-induced fault injection. Additionally, statistical fault injection was performed on a hardware-emulated POWER6 processor simulation model. The error resiliency is described in terms of the proportion of latch upset events that result in vanished errors, corrected errors, checkstops, and incorrect architected states.

System Resilience: Need

Figure 2

POWER6 test system mounted in beamline.

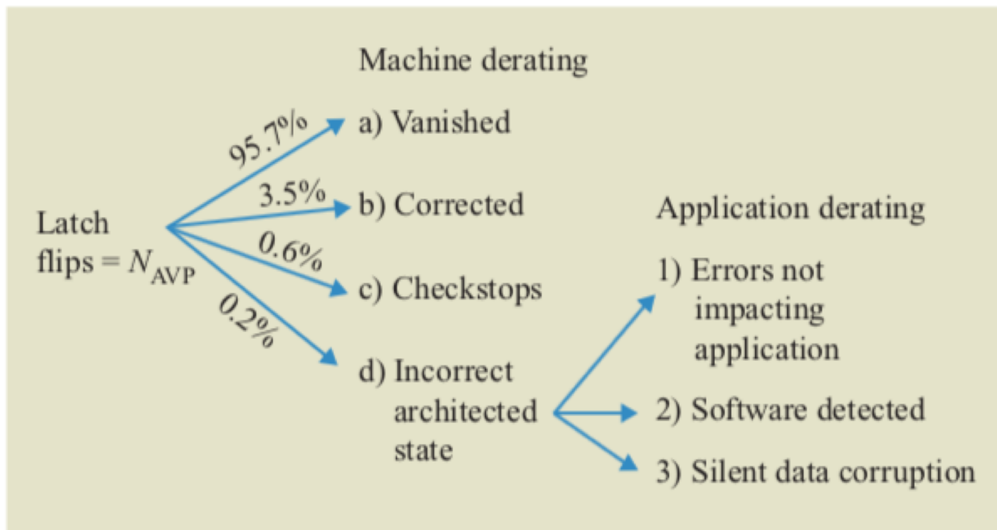


Figure 1

Taxonomy of derating terms.

System Resilience: Need

ORNL/TM-2016/687

Resilience Design Patterns

A Structured Approach to Resilience at Extreme Scale

ORNL Technical Report - Version 1.0

Dec 2016

10 Case Study: Proactive Process Migration

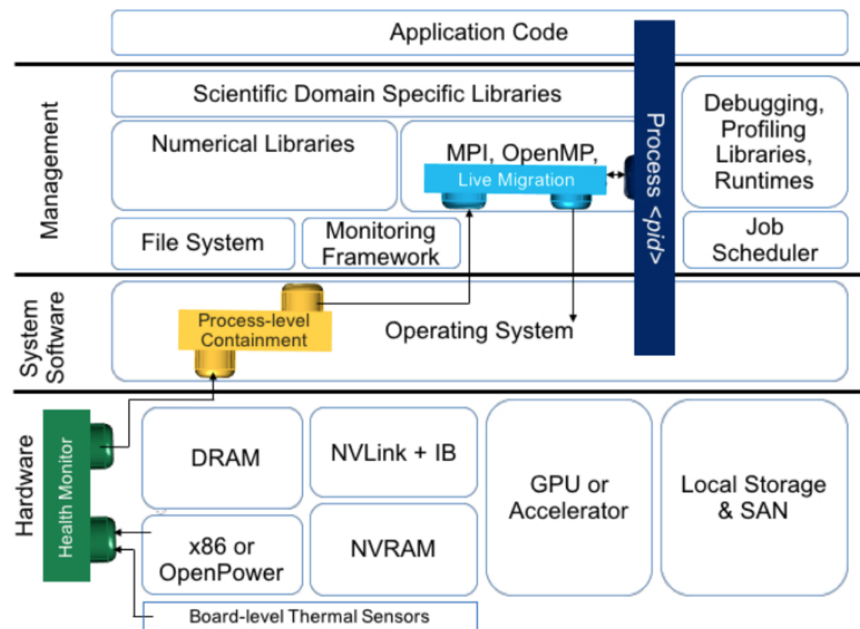
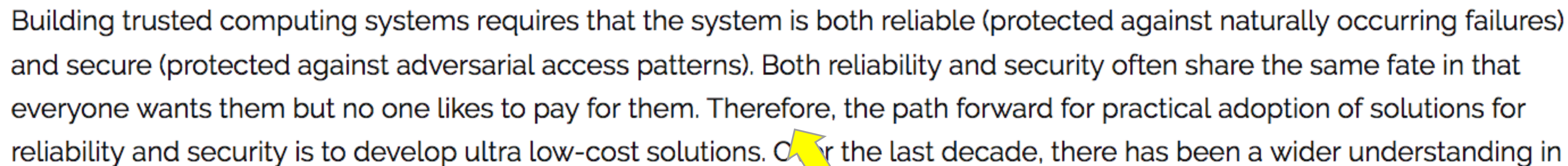


Figure 7: Resilience Solution Case Study: Process Migration

System Resilience: Want

Speaker: Moin Qureshi, Georgia Tech

SELSE 2019


On Exploiting the Synergy Between Reliability and Security

Building trusted computing systems requires that the system is both reliable (protected against naturally occurring failures) and secure (protected against adversarial access patterns). Both reliability and security often share the same fate in that everyone wants them but no one likes to pay for them. Therefore, the path forward for practical adoption of solutions for reliability and security is to develop ultra low-cost solutions. O[illegible]r the last decade, there has been a wider understanding in

System Resilience: Plausible Reasons for Lack of Adoption

- No continued investment (in many cases)
- Community unprepared to stomach costs
  - New fault models -> accepted!
  - What to do after detection -> accepted!
  - Papers on detection itself often rejected
    - as indicated by rejection (plus the stated reasons)
- Nobody wants 30% overhead
  - "Why not go back to earlier lithography?"
- Other problems that make it worse:
  - Lack of guarantees on detection
  - High false positive rates
    - Unacceptable, given that bit-flips themselves are rare!

Path Forward

- Ultra-low costs
- Ultra-tight guarantees

This talk

- Approach to detect with rigorous guarantees
  - Focus on specific domains
    - Stencil codes
  - Offer rigorous guarantees and reasonable overheads
    - Our approach: FPDetect
- Approach to amplify failures
  - To manifest them more
  - Leads to cheaper detection
  - FailAmp

This talk

- Approach to detect with rigorous guarantees
    - Focus on specific domains
        - Stencil codes
    - Offer rigorous guarantees and reasonable overheads
        - Our approach: FPDetect
- Approach to amplify failures
    - To manifest them more
    - Leads to cheaper detection
        - Our approach: FailAmp

This talk

- Approach to detect with rigorous guarantees
  - Focus on specific domains
    - Stencil codes
  - Offer rigorous guarantees and reasonable overheads
    - Our approach: FPDetect
- Approach to amplify failures
  - To manifest them more
  - Leads to cheaper detection
    - Our approach: FailAmp
- Capitalize on custom fault models to obtain lower overheads
- Concluding Remarks:
  - How to ensure that the area stays viable?

FPDetect

- Stencil codes are a good target for protection
    - Higher computational intensity
    - SDCs can build up
        - based on the nature of the PDEs being solved
- Problem with putting assertions around data
    - Don't know exact invariants
    - Machine-learned models tried → too imprecise
        - Weaker invariants will trigger false alarms
- Obvious insight
    - There is an ever-present invariant
        - A duplicated computation!

FPDetect

- Doing duplication naively is unwise
  - Too much overhead
- Our (rather unusual) approach
  - Find what the value will be T steps later!

FPDetect

- Doing duplication naively is unwise
  - Too much overhead
- Our (rather unusual) approach
  - Find what the value will be T steps later!

FPDetect Approach (higher level)

- Find out what the value will be T steps later
- Guarantee b bits of mantissa exactly
  - If at runtime we observe b bits not being preserved, then....
    - Conclude that a bit-flip occurred!

FPDetect Approach

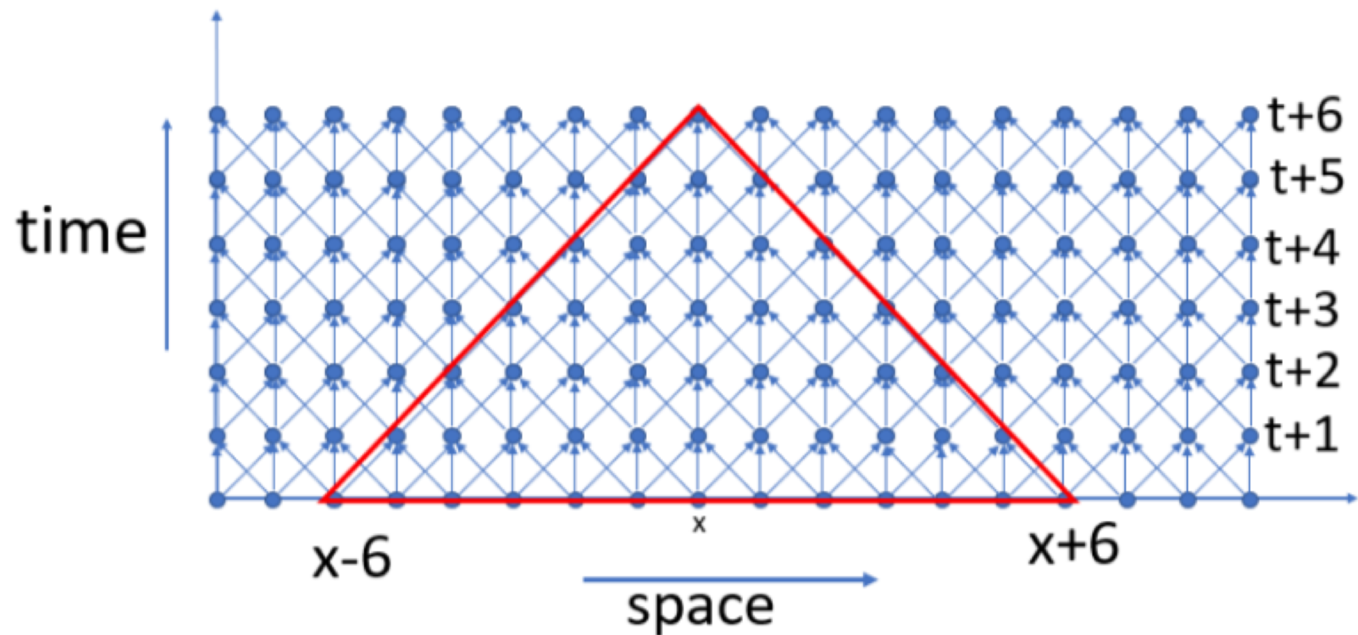


Fig. 1. Simplified 1D stencil over 6 time steps

FPDetect Optimization

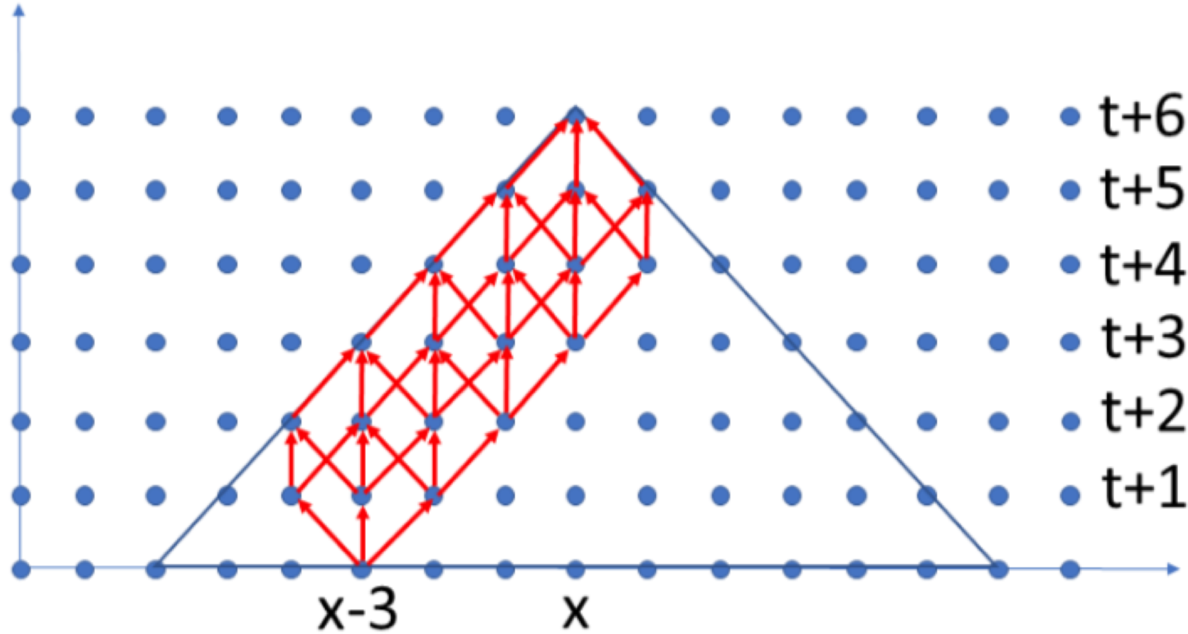


Fig. 2. Illustration of path dominance

Compute per
Binade-difference group
And have it in a table
For lookup

FPDetect Detector Stacking (shows spatial stacking, temporal stacking, and coverage “holes”)

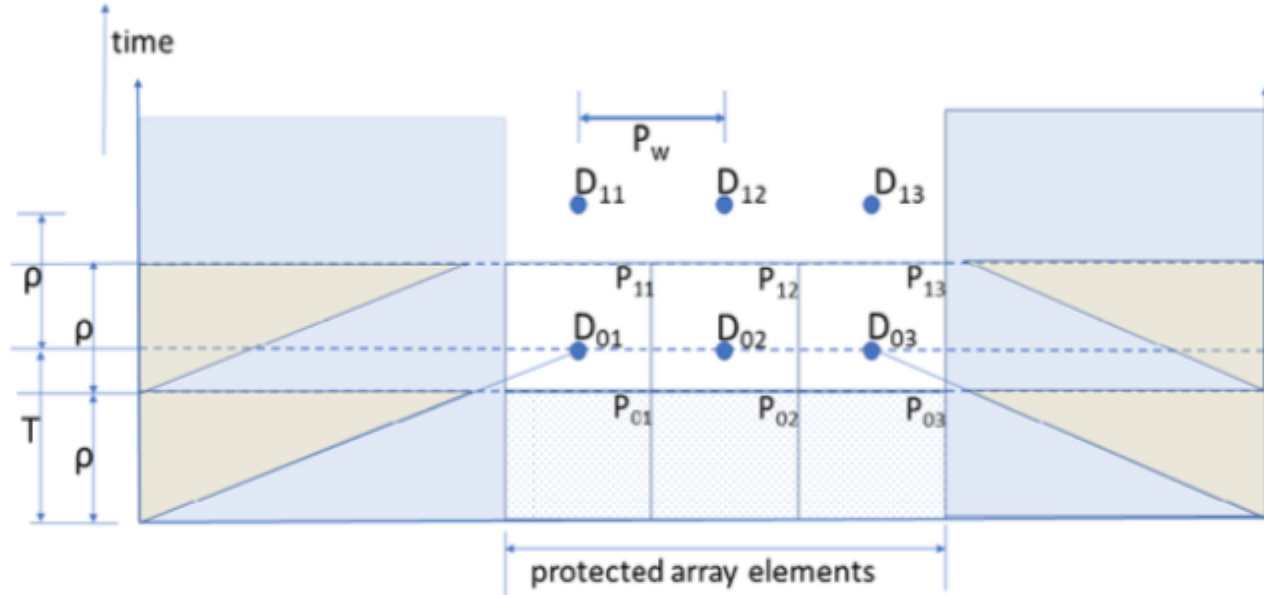
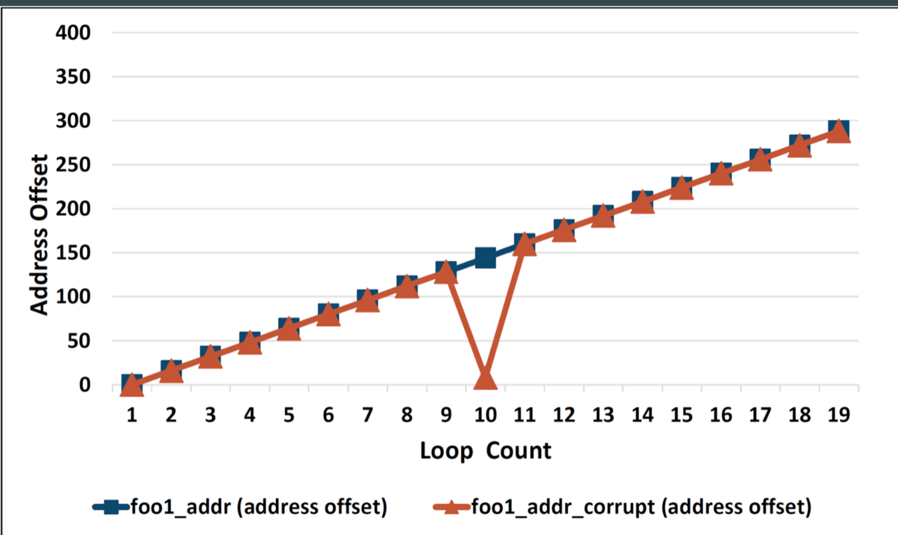


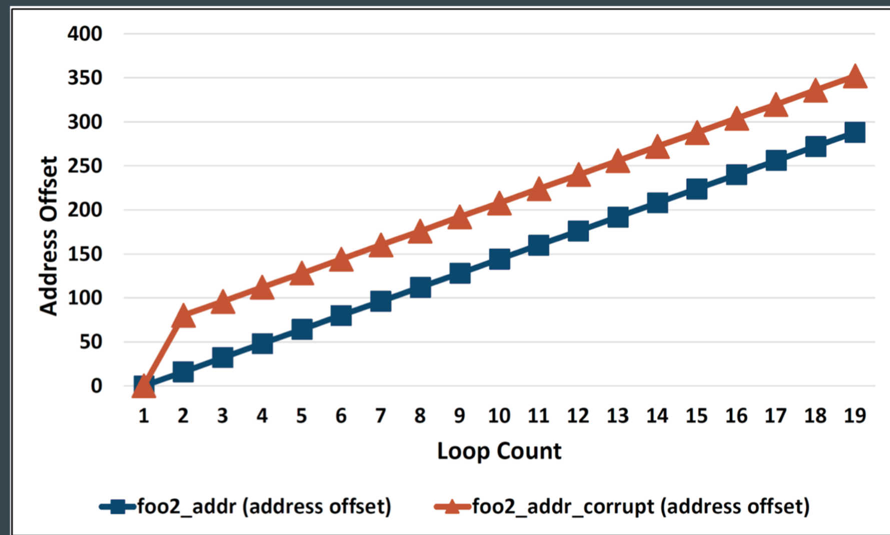
Fig. 6. Detector arrangement for a 1-d stencil. Horizontal: array elements; vertical: time steps

FailAmp

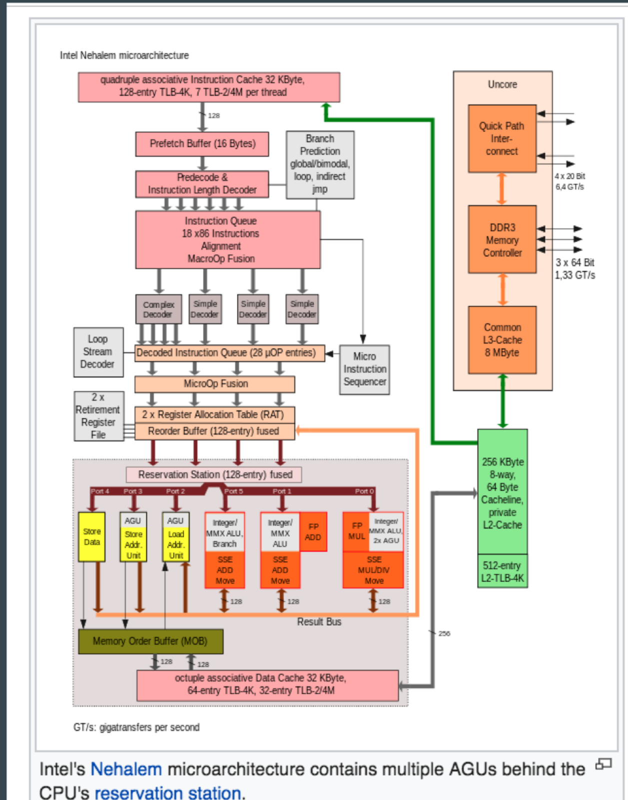
- “Make a bad problem worse”
  - So it can be observed more readily!

FailAmp: Make a transient address "blip" permanent

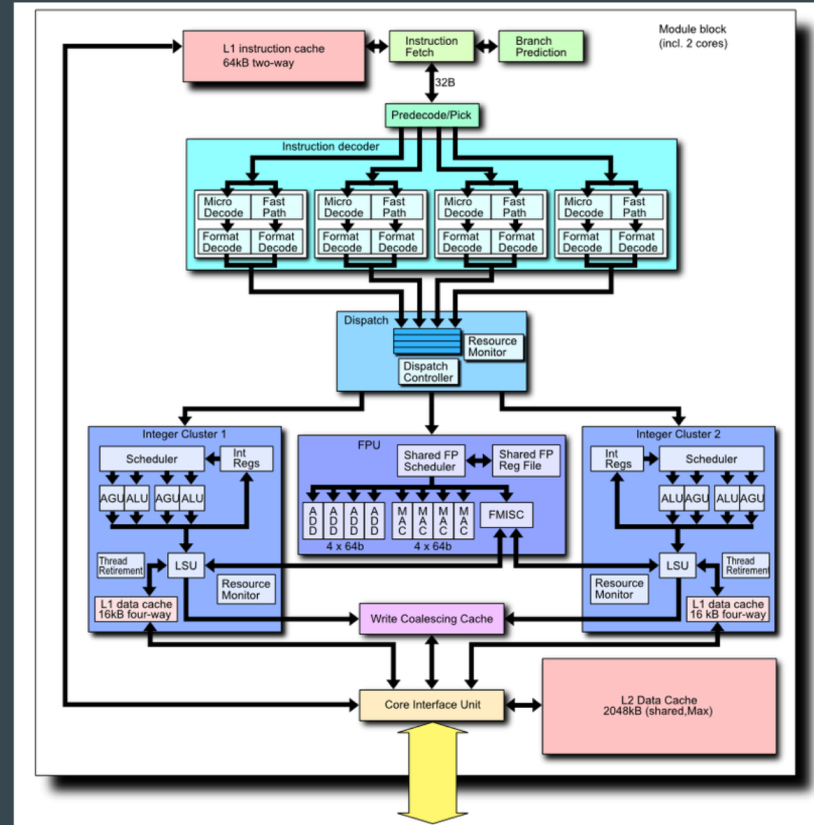




FailAmp protects AGUs (images from Wikipedia below)



Intel's Nehalem microarchitecture contains multiple AGUs behind the CPU's reservation station.



FailAmp in a nutshell

- An LLVM transformation
  - Rewrite the Get Element Pointer instructions pertaining to array accesses
  - Flow relativized addresses via new Phi-nodes
  - Put detectors as frugally as possible
- It is a “whole function relativization”
  - Existing compilers often do for one loop
  - They don’t connect-up relativization chains

FailAmp rewrites GEP instructions

GEPs are

- “One stop shopping” for Arrays of Structs of Arrays
- Also handles vectorization

GEP Example*

```
struct RT {
     int A;
     int B[10][20];
     int C;
}
struct ST {
     struct RT X;
     int Y;
     struct RT Z;
}
int *foo(struct ST *s) {
   return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `%ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `size_ty(%ST)`.
3. Compute the index of the `Z` field by adding `size_ty(%RT) + size_ty(i32)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `size_ty(i32)` to skip past `A`.
5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
    %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
    ret i32* %arrayidx
}
```

Final answer: ADDR + `size_ty(%ST) + size_ty(%RT) + size_ty(i32) + size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)`

*adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

FailAmp rules, and a generic example

$pm[A] = 0,\ pm[e_1] = d_{i_1}$

$e_2 = g(e_1, i_1)$:

- $d_{i_2} = pm[e_1] + i_1$
- $\Delta_2 = d_{i_2} - d_{i_1}$
- $r_{e_2} = g(r_{e_1}, \Delta_2)$
- $pm[e_{i_2} \leftarrow d_{i_2}]$

$e_3 = g(e_2, i_2)$:

- $d_{i_3} = pm[e_2] + i_2$
- $\Delta_3 = d_{i_3} - d_{i_2}$
- $r_{e_3} = g(r_{e_2}, \Delta_3)$
- $pm[e_{i_3} \leftarrow d_{i_3}]$

$e_4 = g(e_2, i_3)$:

- $d_{i4} = pm[e_2] + i_3$
- $\Delta_4 = d_{i_4} - d_{i_3}$
- $r_{e_4} = g(r_{e_3}, \Delta_4)$
- $pm[e_{i_4} \leftarrow d_{i_4}]$

$e_5 = g(A, i_4)$:

- $d_{i5} = pm[A] + i_4$
- $\Delta_5 = d_{i_5} - d_{i_4}$
- $r_{e_5} = g(r_{e_4}, \Delta_5)$
- $pm[e_{i_5} \leftarrow d_{i_5}]$

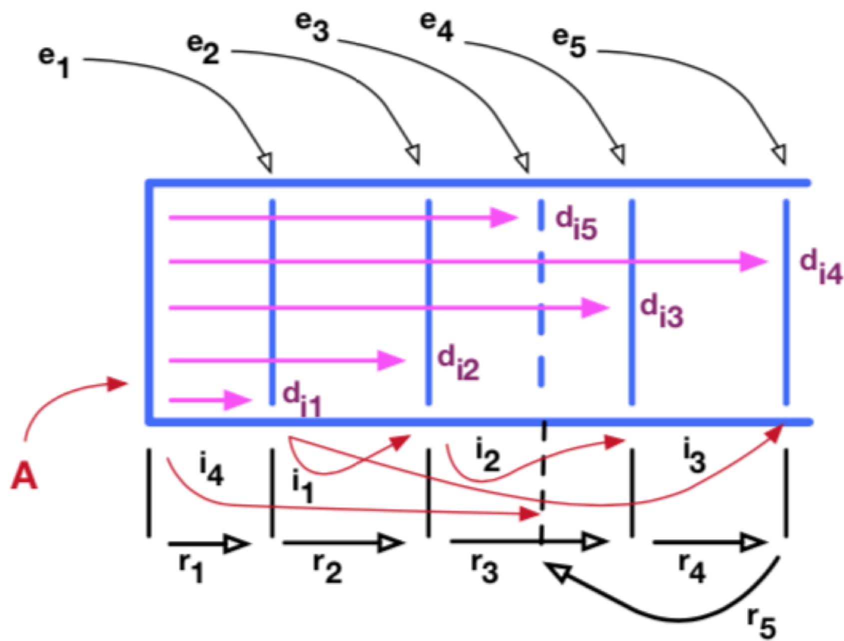


Fig. 2. FailAmp Illustration

FailAmp Compilation Rule (general case)

$$\langle pm,\ \ \%e' = g(\%e, \%i); P,\ \ \%r,\ \ \%d,\ \ R\rangle$$
$$\longrightarrow$$
$$\langle pm+ = (\%r', \%d'),\ \ P,\ \ \%r',\ \ \%d',\ \ R; NewCode\rangle$$

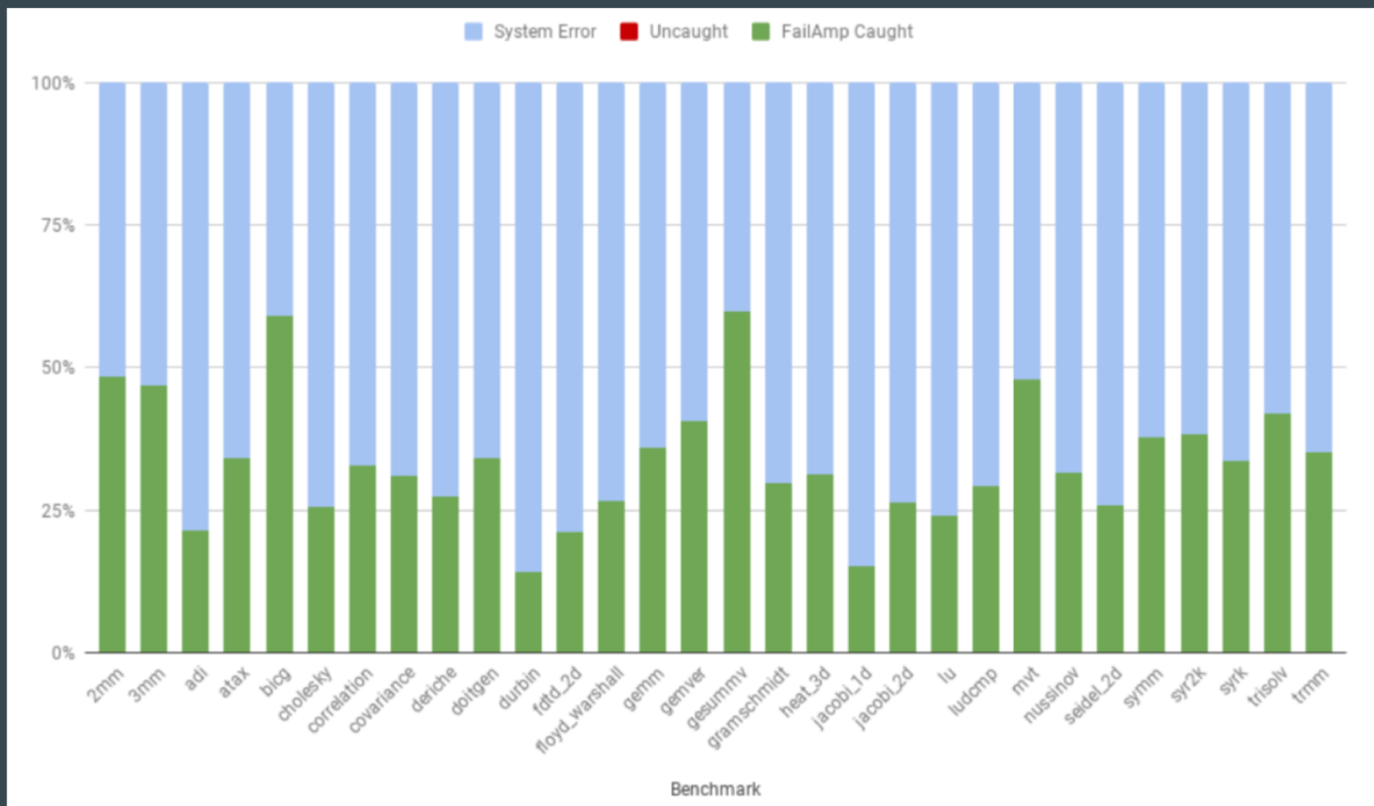
where,

$NewCode =$
$\ \ \%d' = add(pm[\%e], \%i);$
$\ \ \%\Delta = sub(\%d', \%d);$
$\ \ \%r' = g(\%r, \%\Delta)$
$\ \ assert(\%A + \%d' == \%r')$

Fig. 3. Formal Transformation in FailAmp

There are
Special cases
Where the
Generated code
Can be simplified

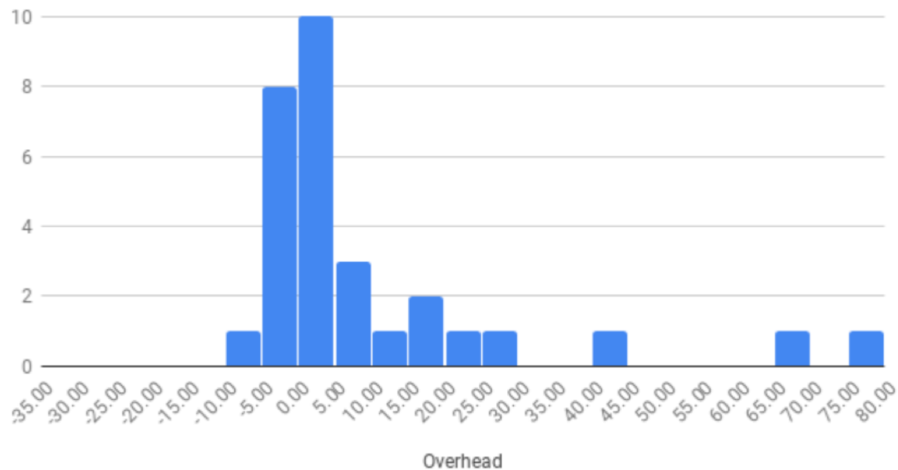
FailAmp Coverage Results

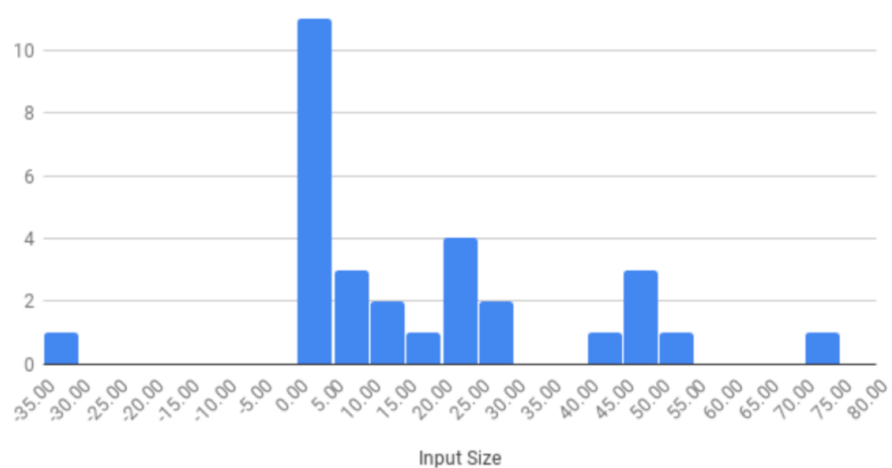


FailAmp highlights

- Found mistake in initial rules
  - Formal verification using SMACK caught mistake
- Now FailAmp catches 100% of all injected address faults
  - Injections done AFTER compiler optimizations (various)
  - This is CRUCIAL to manifest many GEP sequences
- ARM has single instruction that fuses key FailAmp steps
  - Post-indexed addressing
    - Effective Address calculated replaces base address
    - X86 needs 2 instructions (calculate Eff. Addr and load as new base; ARM takes one)
- Preliminary results on LULESH for FailAmp on a 96 x 96 x 96 cube
  - 5% overhead
  - 100% detection of address faults
  - No False Positives!

FailAmp Overhead Results





Concluding Remarks

- We presented FPDetect and FailAMP – two complementary approaches for system resilience
- Both are usable in a context that uses polyhedral optimizations
  - Measured effective before/after PLUTO transformations
- FPDetect also helps detect logical bugs
- Would be interesting to develop interesting mixes of amplification + detection
  - E.g., even FPDetect + FailAmp makes sense...
- Cross-layer resilience schemes are essential to curb overheads and localize faults

- Must view resilience as “End of Moore Insurance”
  - Tight-rope walk at End of Moore
  - Good detectors catch falls and helps us recover

Extra: Intel vs ARM

x86 (Intel syntax):

```
; ebx = current relative pointer
; esi = calculated delta index
; 4 = size of array item
; edx = where data will be loaded into

; calculate new relative pointer
lea ebx, [esi*4 + ebx]
; load pointer
mov edx, ebx
```

ARM:

```
; R2 = current relative pointer
; R3 = just calculated delta index
; LSL 2 = shift applied to delta index (same as mul by 4)
; R1 = where data will be loaded into

; calculate new relative pointer, load value, and replace R2 with new relative pointer
ldr R1, R2, R3, LSL 2
```

Extra: Intel vs ARM

In x86 the normal way to access memory would be:

```
mov edx, [esi*4 + ebx]
```

Which would calculate the pointer and load it, but the `[esi*4 + ebx]` is never placed in a register.
LEA was made to do this calculation and keep the result in a register so that the load can occur later.

ARM was made with the intention of modifying a pointer on access, so we get the new pointer without having to 'split' instructions.