Approximate Computing: (Old) Hype or New Frontier?

Sarita Adve

University of Illinois, EPFL

Acks: Vikram Adve, Siva Hari, Man-Lap Li, Abdulrahman Mahmoud, Helia Naemi, Pradeep Ramachandran, Swarup Sahoo, Radha Venkatagiri, Yuanyuan Zhou

What is Approximate Computing?

- Trading output quality for resource management?
- Exploiting applications' inherent ability to tolerate errors?

Old?

GRACE [2000-05] rsim.cs.illinois.edu/grace

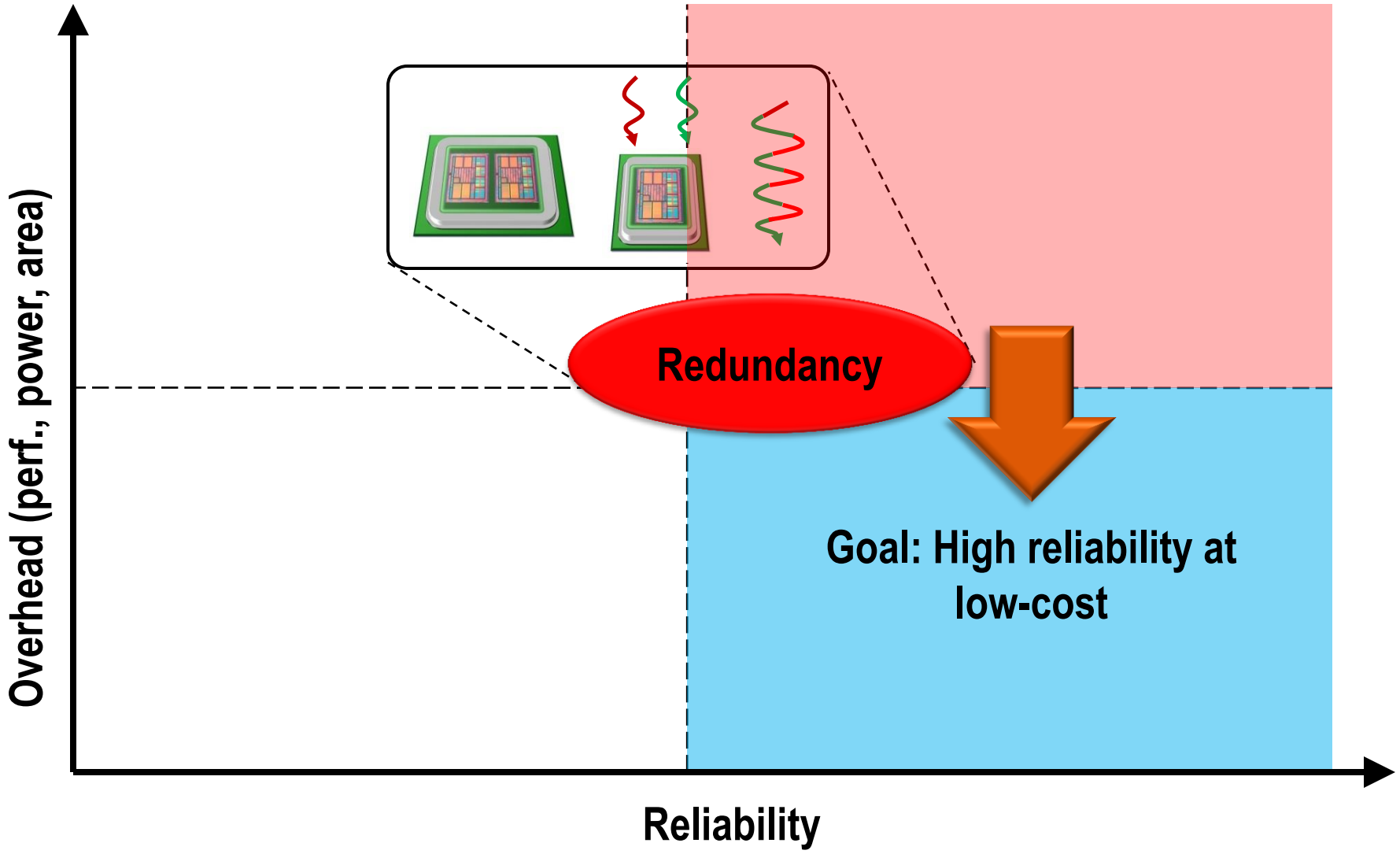
SWAT [2006-] rsim.cs.illinois.edu/swat

Many others

Or something new?

Approximate computing through the lens of hardware resiliency …

Motivation


Overhead (perf., power, area)
Redundancy
Goal: High reliability at low-cost
Reliability

SWAT: A Low-Cost Reliability Solution

- Need handle only hardware faults that affect software

⇒ Watch for software anomalies (symptoms)

  – Zero to low overhead "always-on" monitors

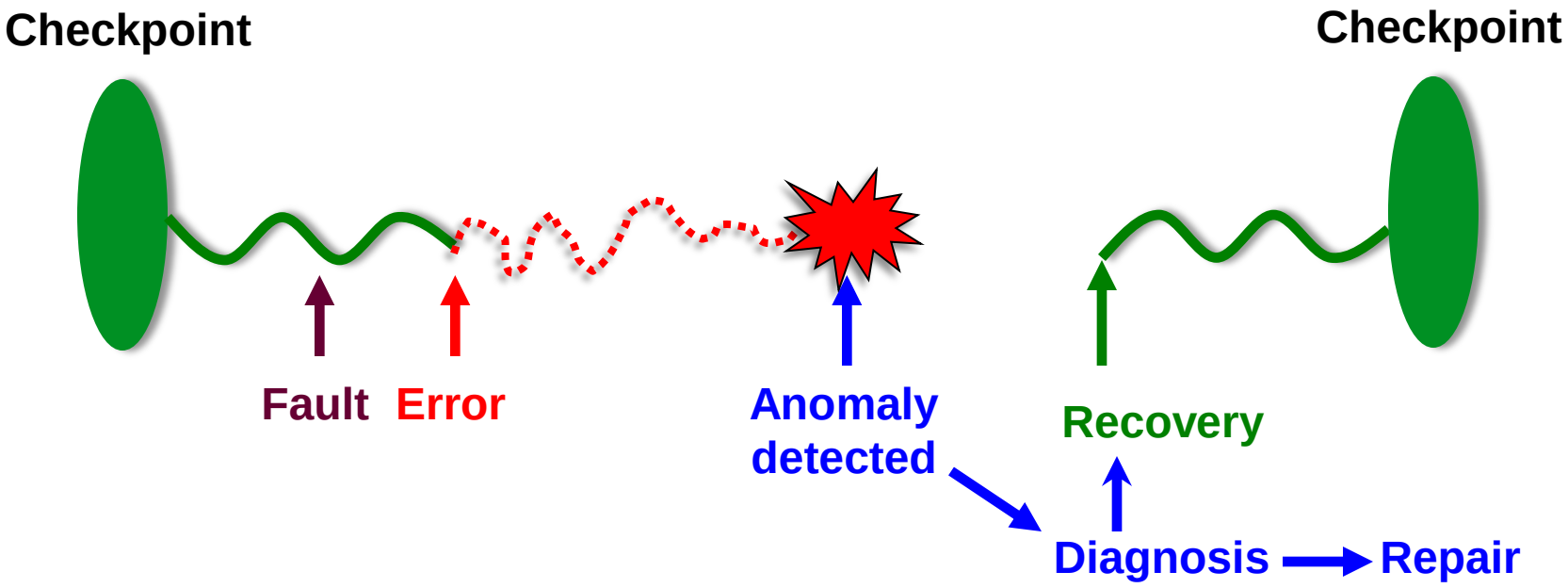
Diagnose cause after anomaly detected and recover

  – May incur high overhead, but invoked infrequently

⇒ SWAT: SoftWare Anomaly Treatment

SWAT Framework Components

- **Detection:** Monitor symptoms of software misbehavior
- **Diagnosis:** Rollback/replay on multicore
- **Recovery:** Checkpoint/rollback or app-specific action on acceptable errors
- **Repair/reconfiguration:** Redundant, reconfigurable hardware
- **Flexible control through firmware**



SWAT Framework Components

- **Detection:** Monitor symptoms of software misbehavior
- **Diagnosis:** Rollback/replay on multicore
- **Recovery:** Checkpoint/rollback or app-specific action on acceptable errors
- **Repair/reconfiguration:** Redundant, reconfigurable hardware
- **Flexible control through firmware**

How to bound silent (escaped) errors?

When is an error acceptable (at some utility) or unacceptable?

How to trade silent errors with resources?

How to associate recovery actions with errors?

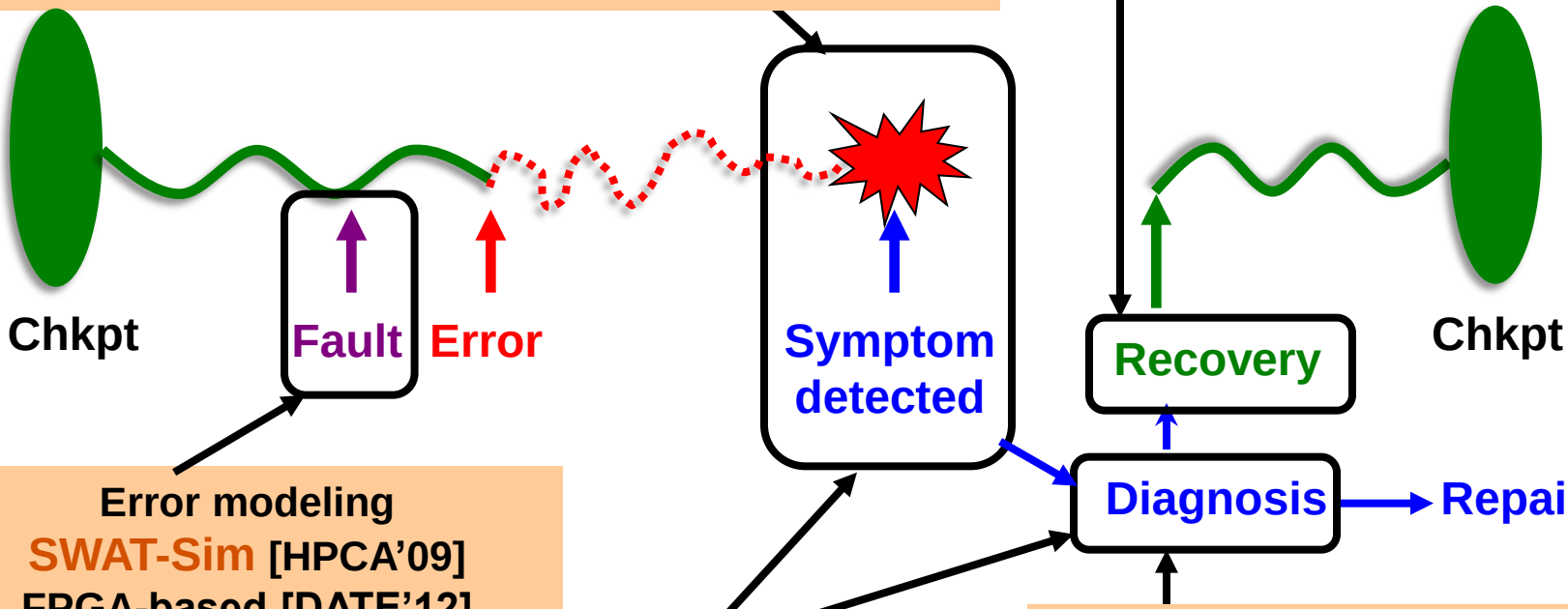
Advantages of SWAT

- **Handles only errors that matter**
  - Oblivious to masked error, low-level failure modes, software-tolerated errors
- **Low, amortized overheads**
  - Optimize for common case, exploit software reliability solutions
- **Customizable and flexible**
  - Firmware control can adapt to specific reliability needs
- **Holistic systems view enables novel solutions**
  - Software-centric synergistic detection, diagnosis, recovery solutions
- **Beyond hardware reliability**
  - Long term goal: unified system (HW+SW) reliability
  - Systematic trade off between resource usage, quality, reliability

SWAT Contributions (for in-core hw faults)

Fault Detection
Very low cost detectors [ASPLOS'08, DSN'08]
Bound silent errors [ASPLOS'12, DSN'12]
Identify error outcomes: Relyzer, GangES [ISCA'14]

Fault Recovery
Handling I/O [TACO'15]



Error modeling
SWAT-Sim [HPCA'09]
FPGA-based [DATE'12]

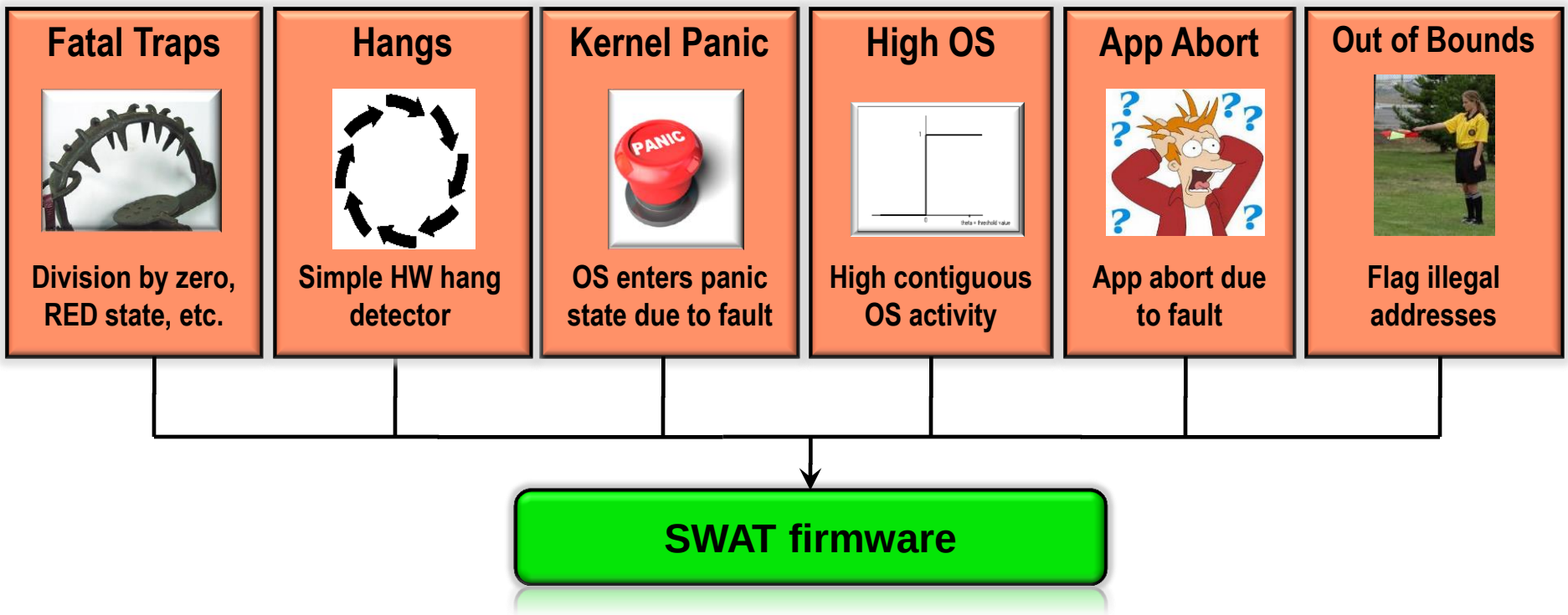
mSWAT [MICRO'09]
Multicore detection & diagnosis

In-situ Diagnosis [DSN'08]
Trace-based µarch diagnosis

Complete solution for in-core faults, evaluated for variety of workloads

SWAT Fault Detection

- Simple monitors observe anomalous SW behavior [ASPLOS'08, MICRO'09]

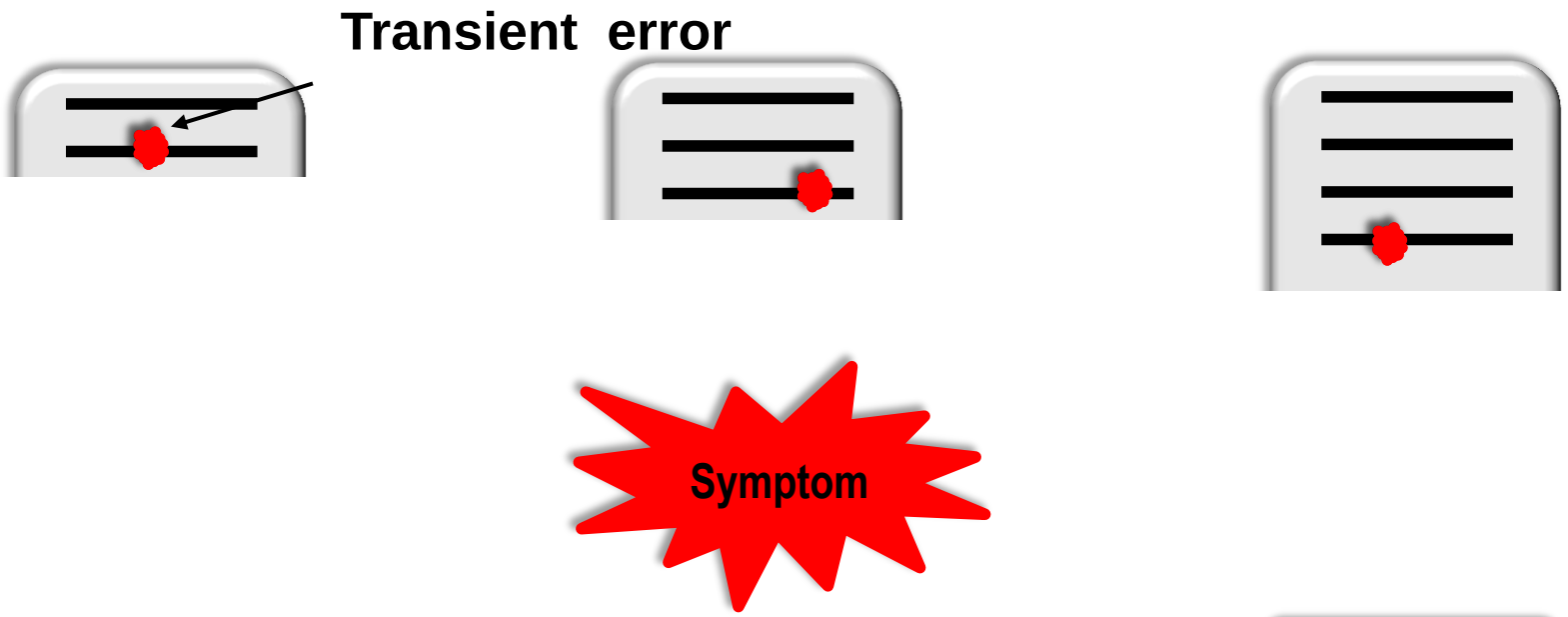


- Very low hardware area, performance overhead

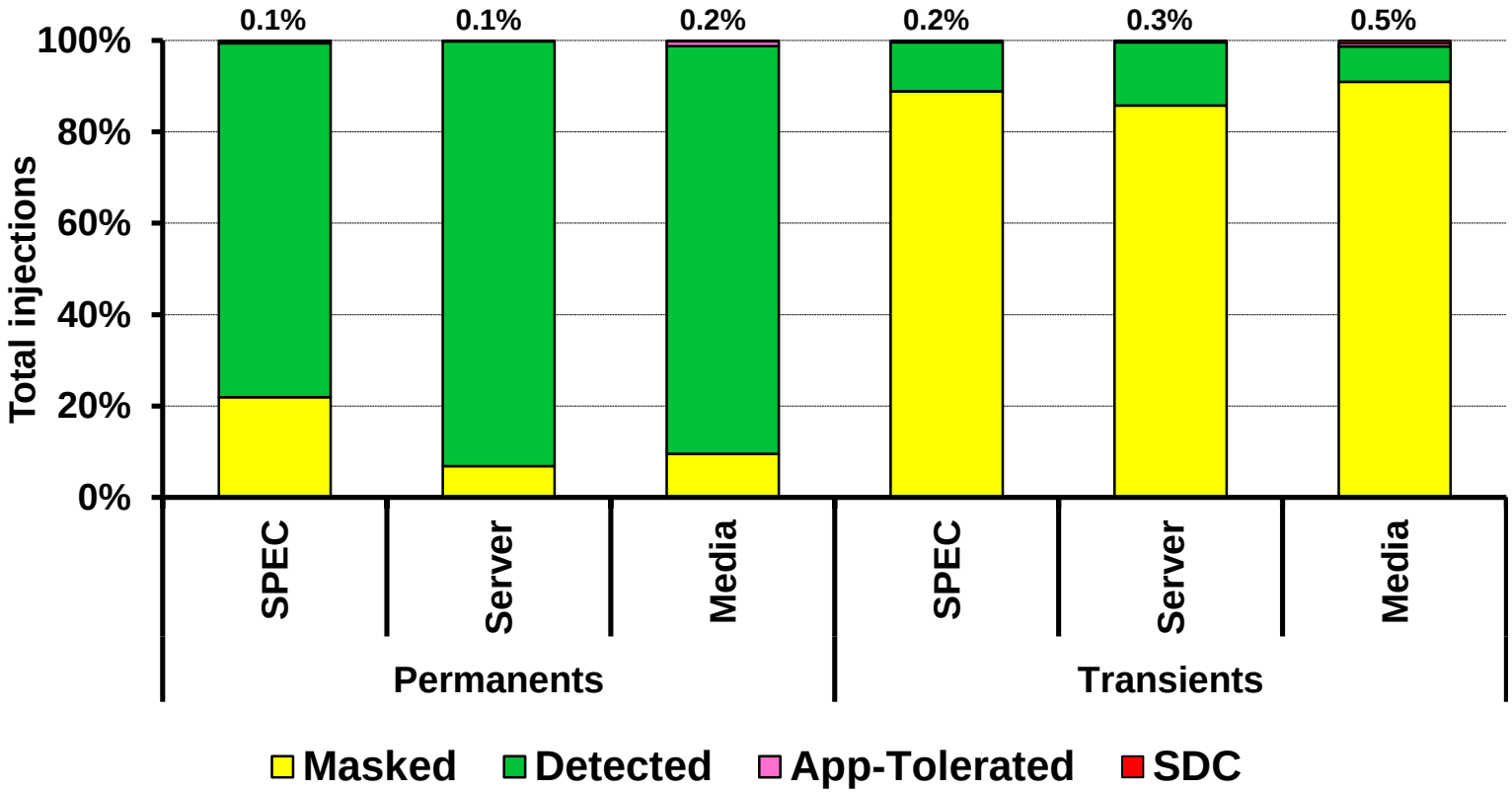
Evaluating SWAT Detectors So Far

- Full-system simulation with out-of-order processor
  - Simics functional + GEMS timing simulator
  - Single core, multicore, distributed client-server
- Apps: Multimedia, I/O intensive, & compute intensive
  - Errors injected at different points in app execution
- µarch-level error injections (single error model)
  - Stuck-at, transient errors in latches of 8 µarch units
  - ~48,000 total errors

Error Outcomes

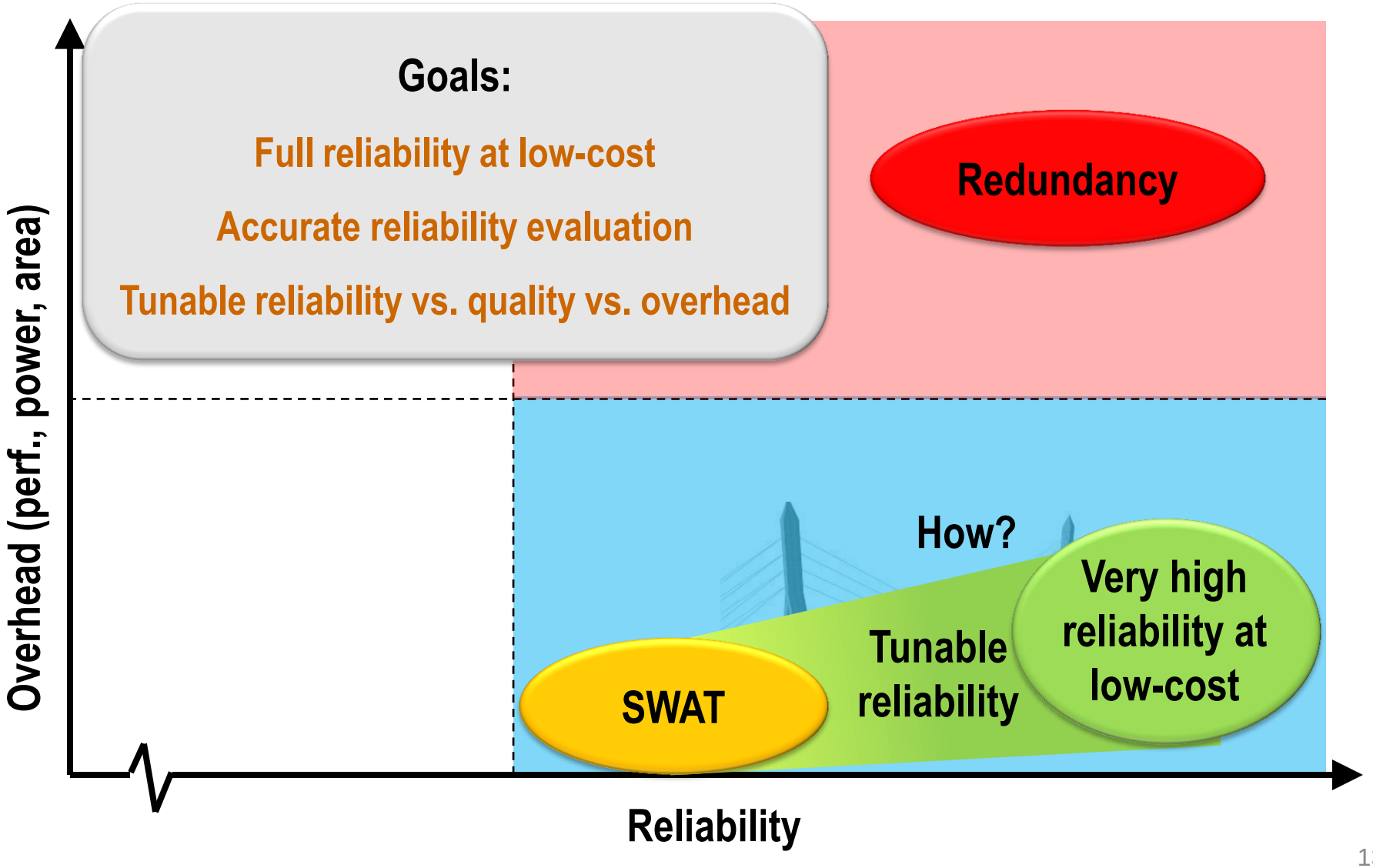


SWAT SDC Rates



- SWAT detectors effective
- <0.2% of injected µarch errors give unacceptable SDCs (109/48,000 total)
- BUT hard to sell empirically validated, rarely incorrect systems

Challenges


Goals:
Full reliability at low-cost
Accurate reliability evaluation
Tunable reliability vs. quality vs. overhead
Redundancy
How?
SWAT
Tunable reliability
Very high reliability at low-cost
Overhead (perf., power, area)
Reliability

Research Strategy

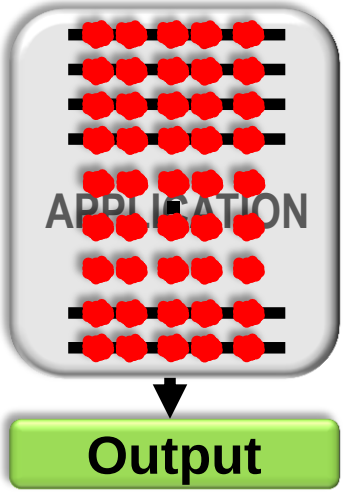
- Towards an Application Resiliency Profile
  - For a given instruction what is the outcome of an error?
  - For now, focus on a transient error in single bit of register
- Convert SDCs into (low-cost) detections for full reliability and quality
- OR let some acceptable or unacceptable SDCs escape
  - Quantitative tuning of reliability vs. overhead vs. quality

Challenges and Approach

- **Determine error outcomes for *all* application-sites**

How?

Complete app resiliency evaluation



Impractical, too many injections

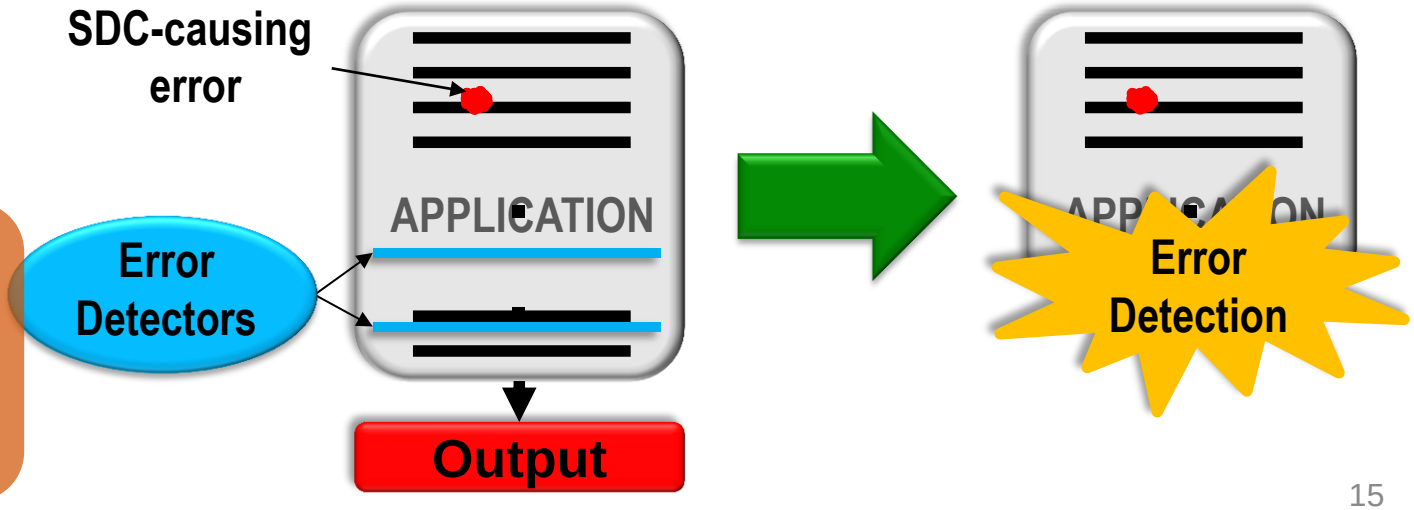
>1,000 compute-years for one app

Challenge: Analyze all errors with few injections

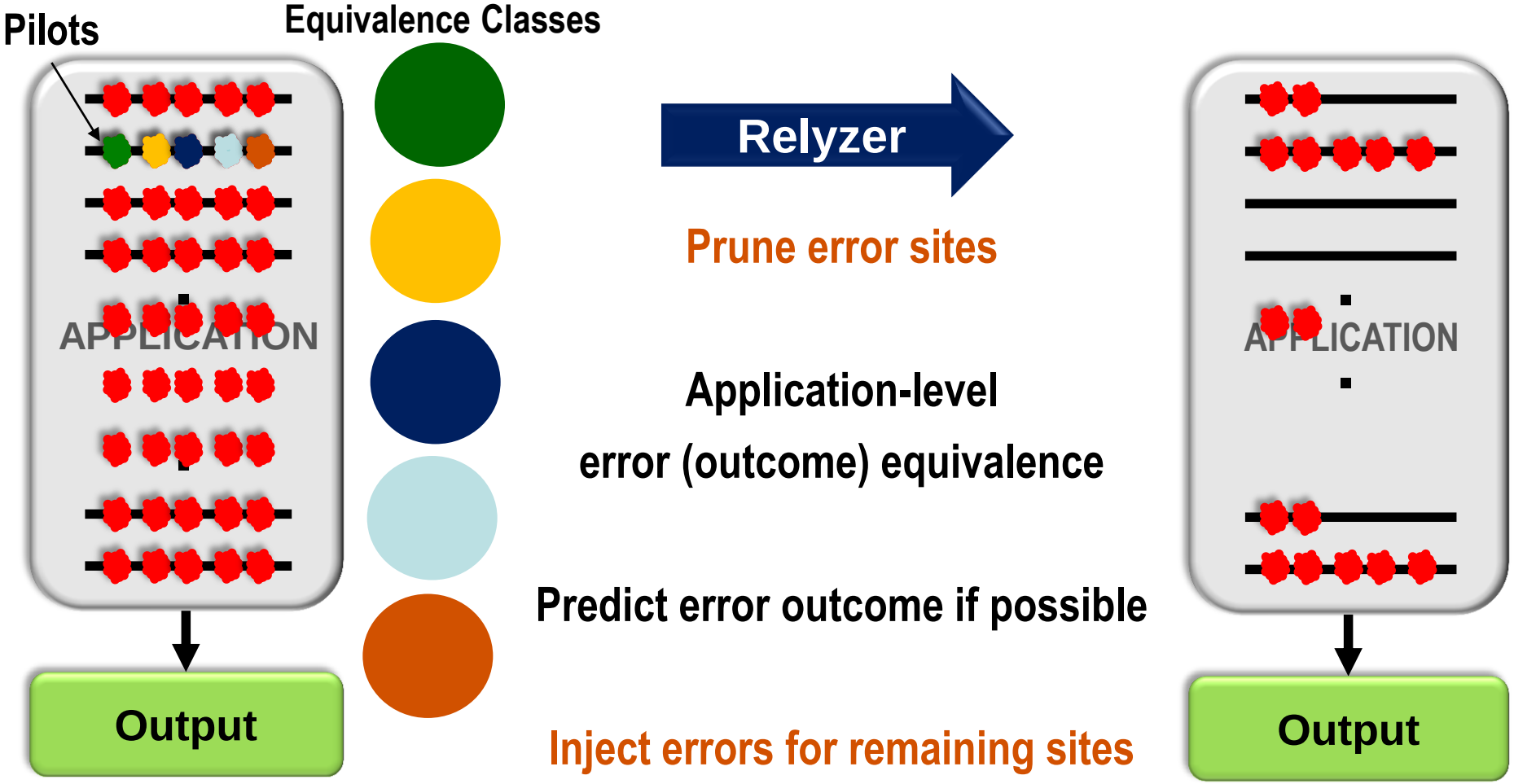
- **Cost-effectively convert (some) SDCs to Detections**

How?

Challenges:
What to use?
Where to place?
How to tune?



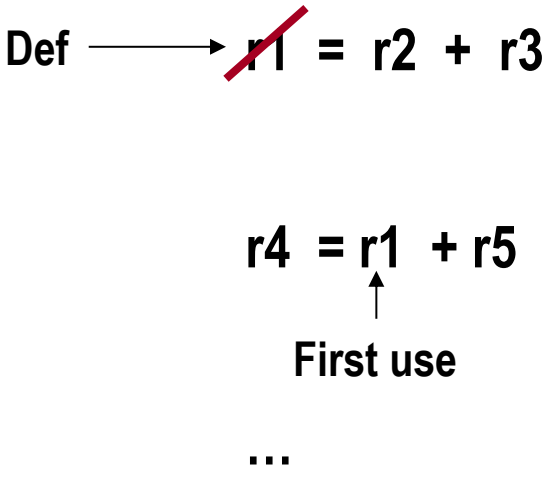
Relyzer: Application Resiliency Analyzer [ASPLOS'12]



Can list virtually all SDC-causing instructions

Def to First-Use Equivalence

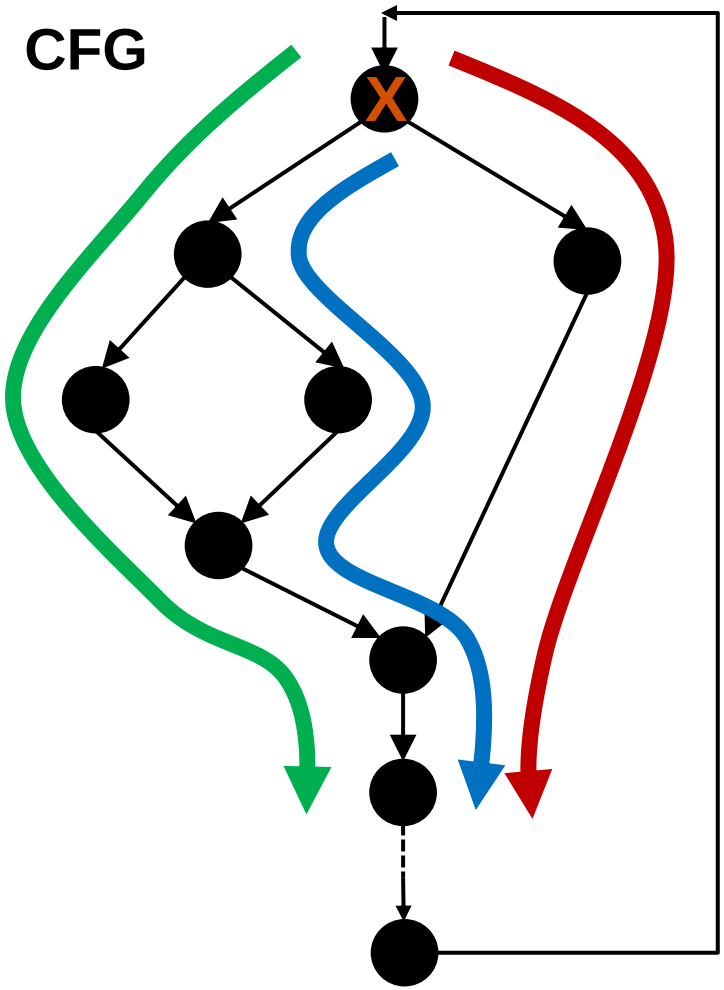
- Fault in first use is equivalent to fault in def ⇒ prune def



- If there is no first use, then def is dead ⇒ prune def

Control Flow Equivalence

Insight: Errors flowing through similar control paths may behave similarly

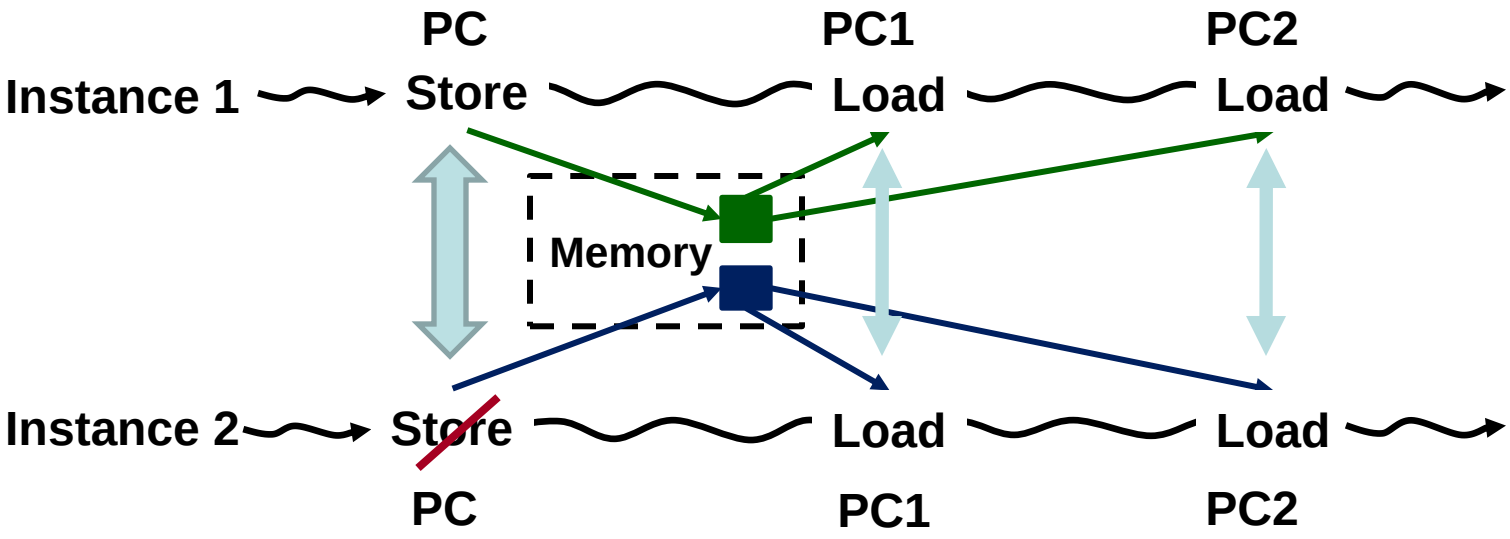


Errors in X that take ■ path behave similarly

Heuristic: Use direction of next 5 branches

Store Equivalence

- **Insight: Errors in stores may be similar if stored values are used similarly**
- **Heuristic to determine similar use of values:**
  - **Same number of loads use the value**
  - **Loads are from same PCs**



Relyzer

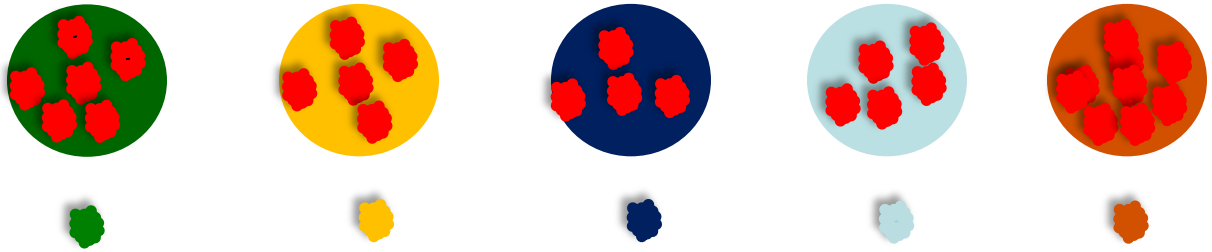
- Relyzer: A tool for complete application resiliency analysis
  - Employs systematic error pruning using static & dynamic analysis
  - Currently lists outcomes of masked, detection, SDC

- 3 to 6 orders of magnitude fewer error injections for most apps
  - 99.78% error sites pruned, only 0.004% sites represent 99% of all sites
  - GangES speeds up error simulations even further [ISCA'14]
- Can identify virtually all SDC causing error sites
- What about unacceptable vs. acceptable SDC? Ongoing

Can Relyzer Predict if SDC is Acceptable?

Equivalence Classes

Pilots



- Pilot = Acceptable SDC ⇒

  All faults in class = Acceptable SDCs ???

- Pilot = Acceptable SDC with quality Q ⇒

  All faults in class = Acceptable SDCs with quality Q ???

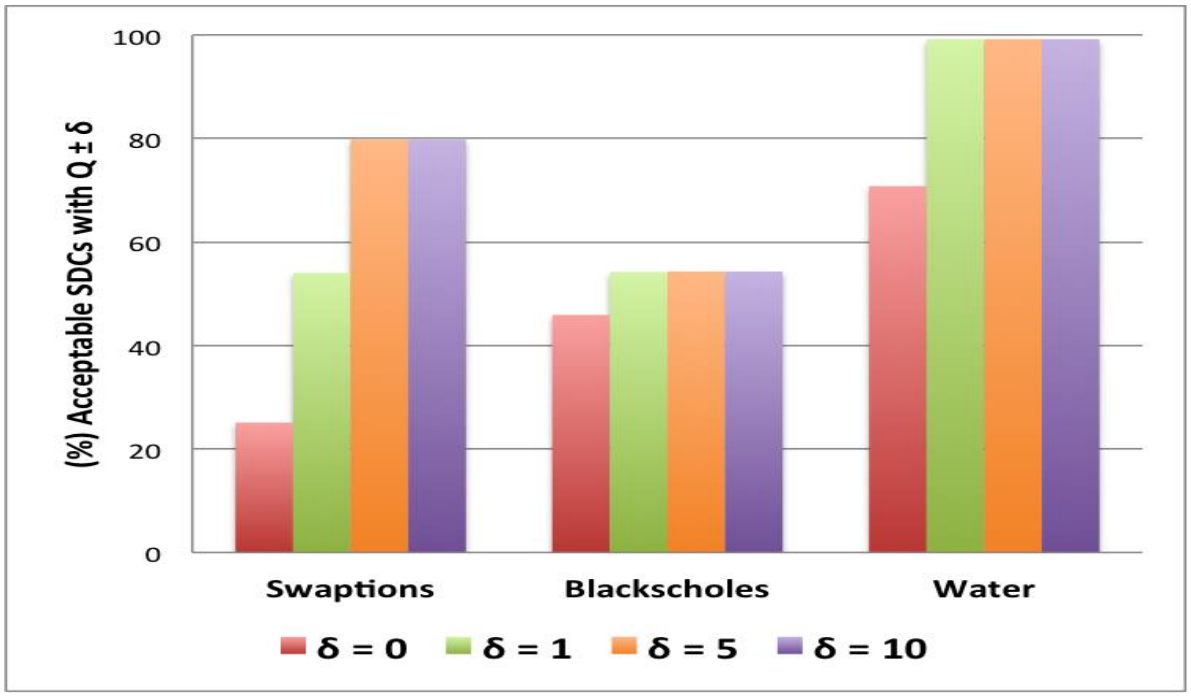
PRELIMINARY Results for Utility Validation

- Pilot = Acceptable SDC with quality Q ⇒

  All faults in class = Acceptable SDCs with quality Q ± δ ???

- Studied several quality metrics
  - E.g., E = abs(percentage average relative error in output components), capped to 100%

Q = 100 - ⌈E⌉ (> 100% error gives Q=0, 1% gives Q=99)



Research Strategy

- Towards an Application Resiliency Profile
  - For a given instruction what is the outcome of a fault?
  - For now, focus on a transient fault in single bit of register
- Convert SDCs into (low-cost) detections for full reliability and quality
- OR let some acceptable or unacceptable SDCs escape
  - Quantitative tuning of reliability vs. overhead vs. quality

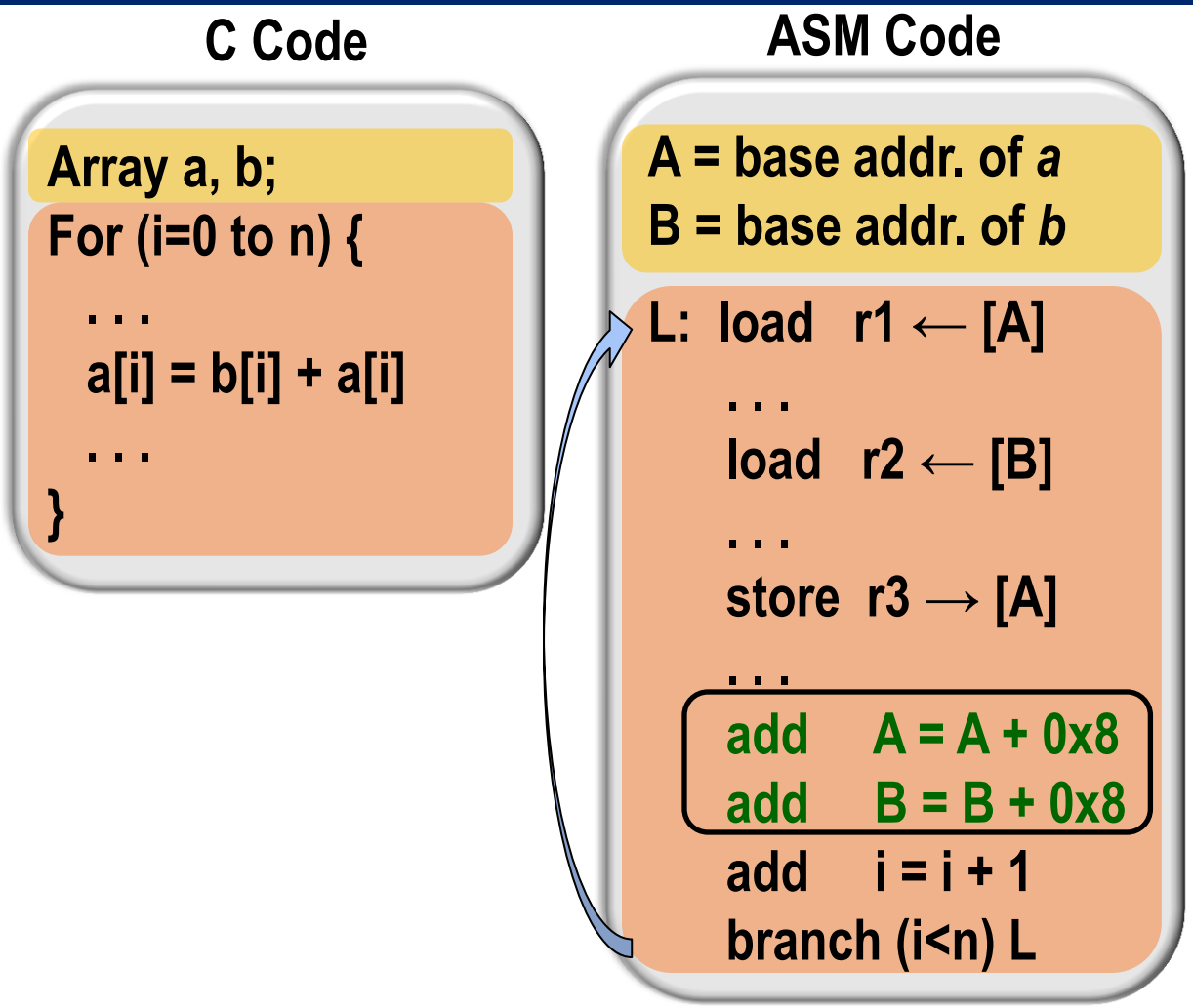
SDCs → Detections [DSN'12]

<table>
<tr><td>What to protect?</td><td colspan="2">SDC-causing fault sites</td></tr>
<tr><td rowspan="4">How to Protect?</td><td colspan="2">Low-cost detectors</td></tr>
<tr><td>Where to place?</td><td>Many errors propagate to few program values</td></tr>
<tr><td>What detectors?</td><td>Program-level properties tests</td></tr>
<tr><td>Uncovered fault-sites?</td><td>Selective instruction-level duplication</td></tr>
</table>

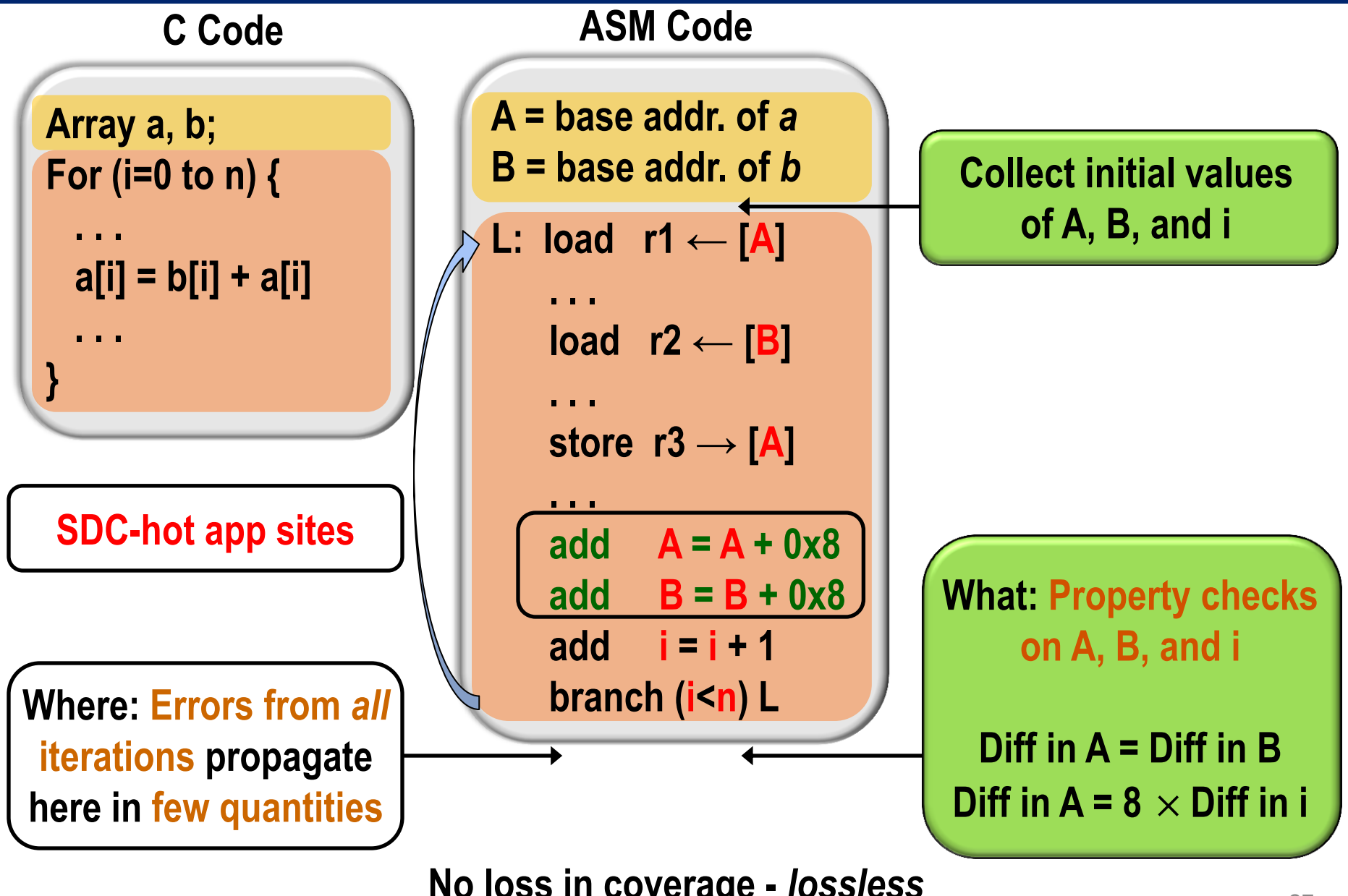
Insights for Program-level Detectors

- Goal: Identify *where* to place the detectors and *what* detectors to use
- Placement of detectors (*where*)
  - Many errors propagate to few program values
    - End of loops and function calls
- Detectors (*what*)
  - Test program-level properties
    - E.g., comparing similar computations and checking value equality

Loop Incrementalization

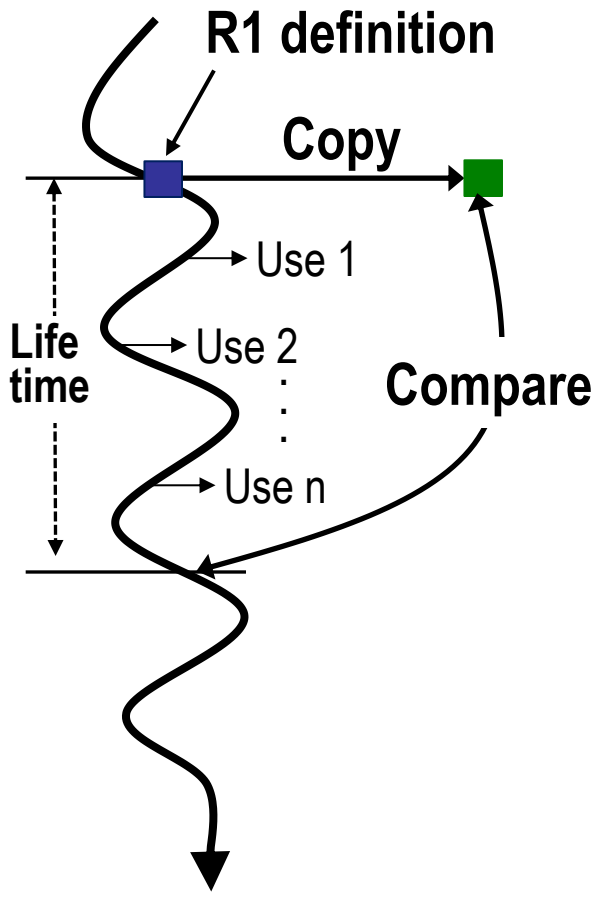


Loop Incrementalization



Registers with Long Life

- Some long lived registers are prone to SDCs
- For detection
  - Duplicate the register value at its definition
  - Compare its value at the end of its life
- No loss in coverage - *lossless*

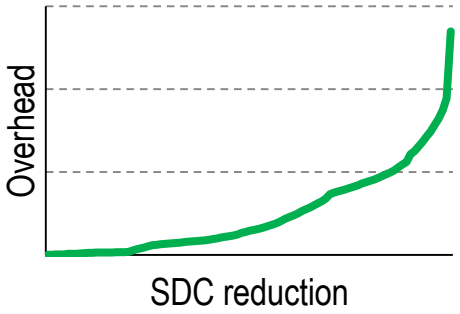


Converting SDCs to Detections: Results

- Discovered common program properties around most SDC-causing sites

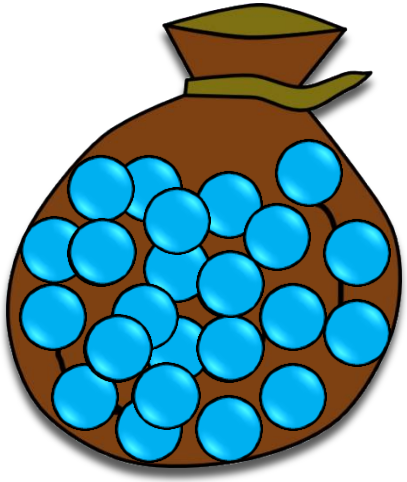
- Devised low-cost program-level detectors
  - Average SDC reduction of 84%
  - Average cost of 10%

- New detectors + selective duplication = Tunable resiliency at low-cost
  - Found near optimal detectors for any SDC target



Identifying Near Optimal Detectors: Naïve Approach

Example: Target SDC coverage = 60%



Bag of detectors

Overhead = 10%

Sample 1

SFI

SDC coverage

50%

Overhead = 20%

Sample 2

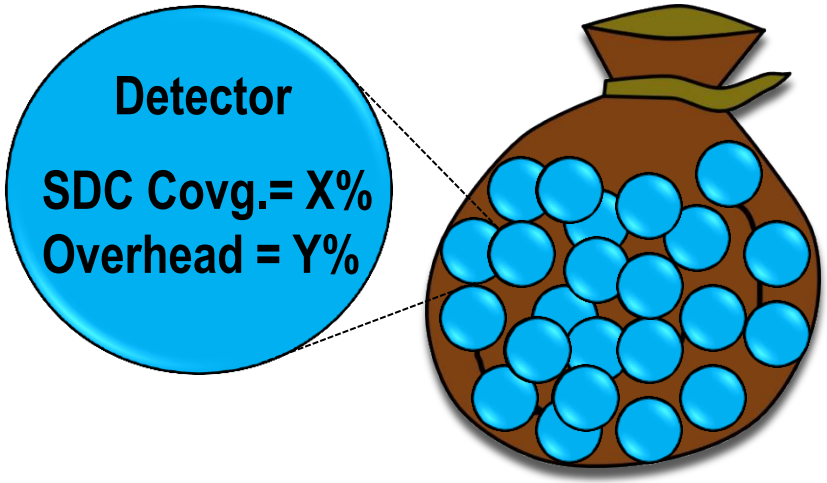
SFI

65%

Tedious and time consuming

Identifying Near Optimal Detectors: Our Approach

1. Set attributes, enabled by Relyzer



Bag of detectors

2. Dynamic programming

Constraint: Total SDC covg. ≥ 60%

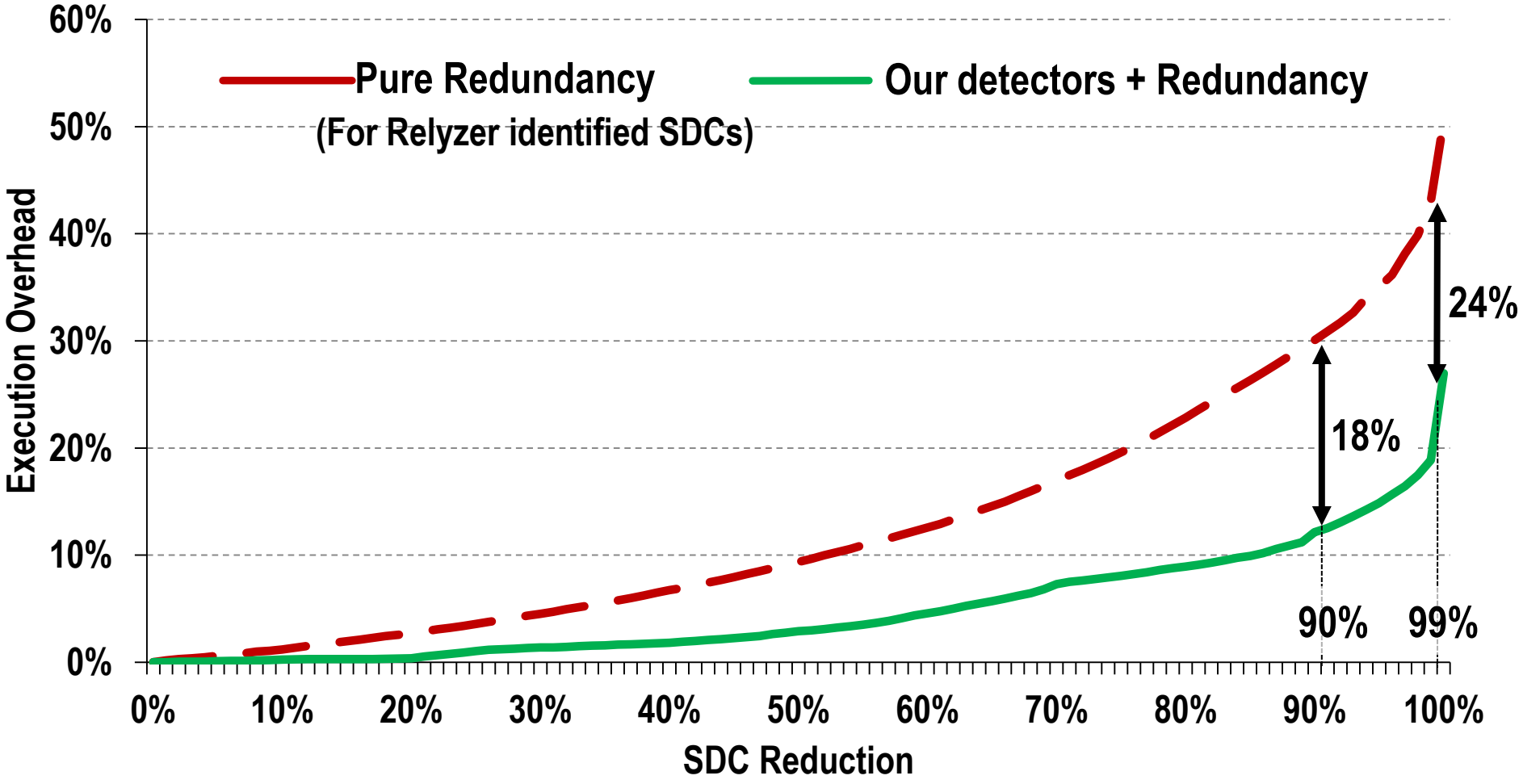
Objective: Minimize overhead

Selected Detectors

Overhead = 9%

Obtained SDC coverage vs. Performance trade-off curves [DSN'12]

SDC Reduction vs. Overhead Trade-off Curve

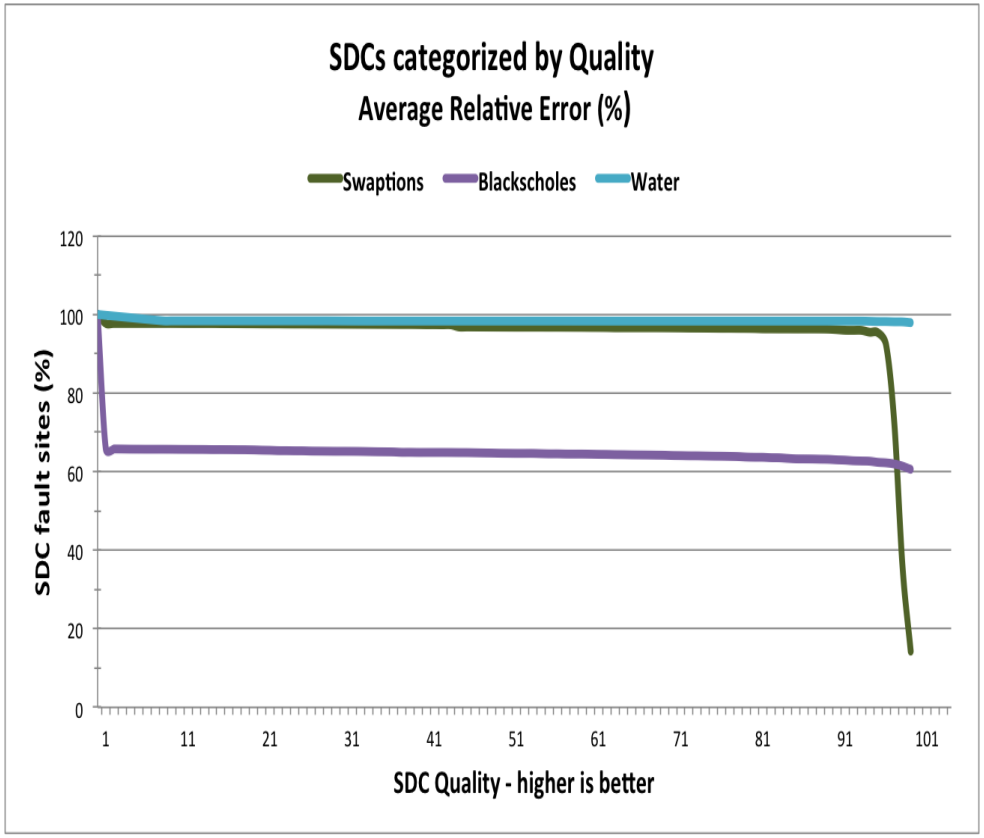


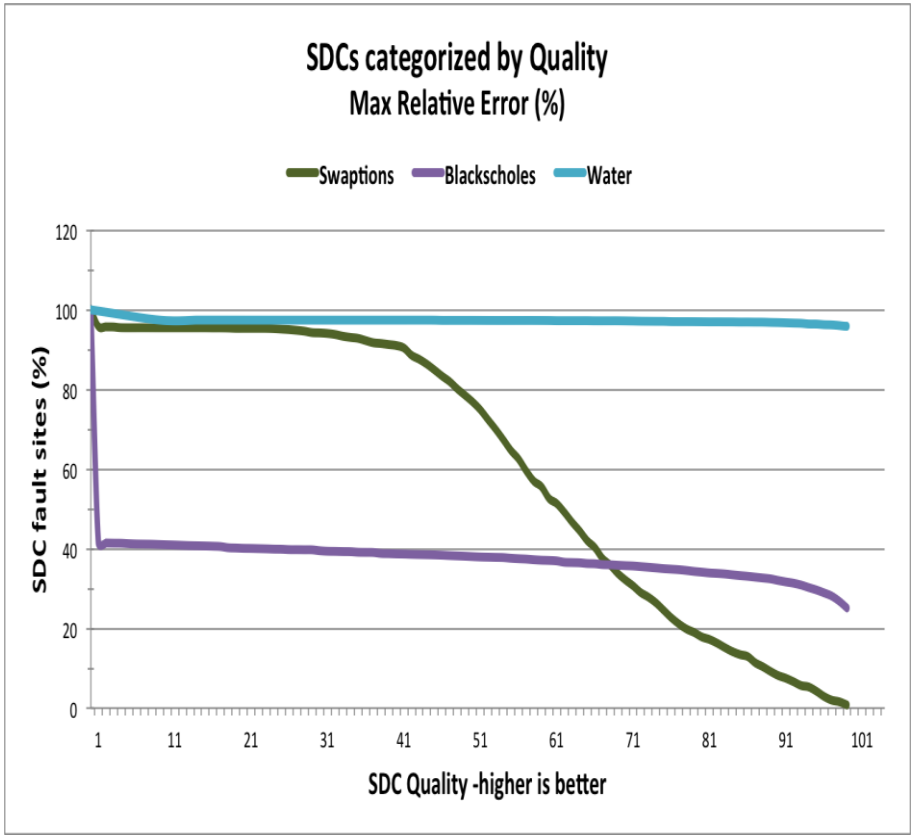
Consistently better over pure instruction-level duplication (w/ Relyzer)

But overhead still significant for very high resilience

Remove protection overhead for acceptable (bounded) quality loss?

Understanding Quality Outcomes with Relyzer (Prelim Results)

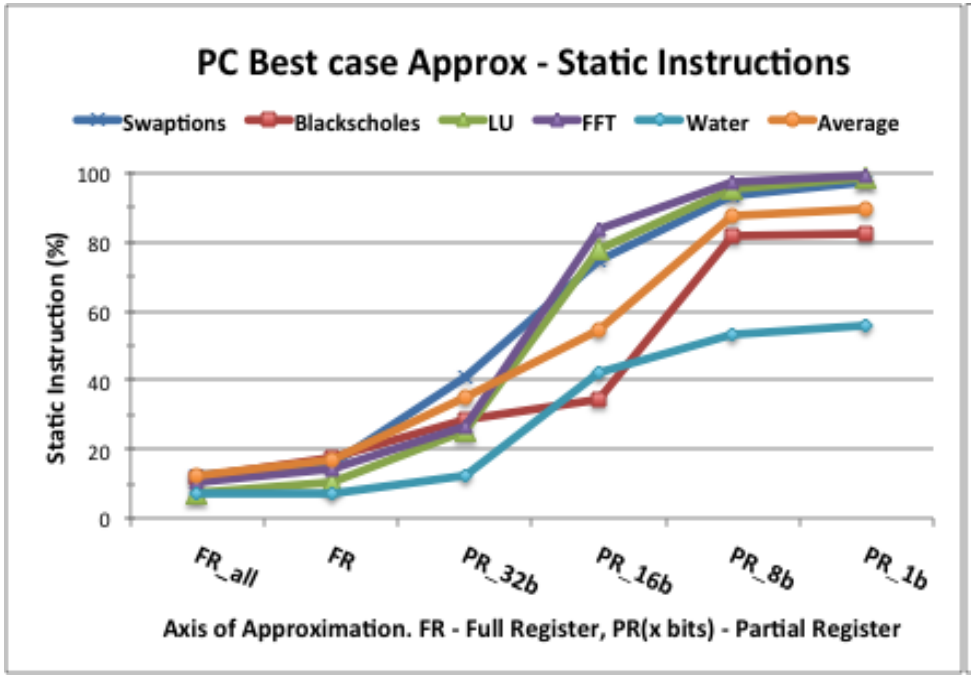


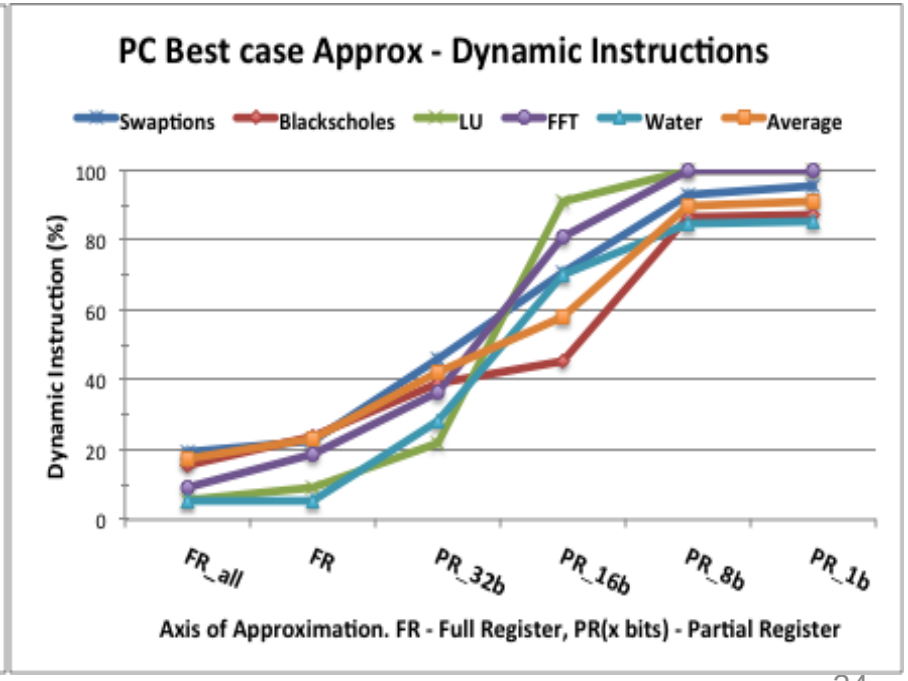


- **Promising potential, but quality metric and application dependent**
- **Can use to quantitatively tune quality vs. reliability vs. resources**

Quality Outcomes – An Instruction-Centric View

- Systems operate at higher granularity than error sites: Instruction?
- When is an instruction approximable?
  - Which errors in instruction should result in acceptable outcome?
    - All? Single-bit errors in all register bits? Single-bit errors in subset of bits?
  - When is an outcome acceptable?
    - Best case : all errors that are masked or produce SDCs are acceptable





So - (Old) Hype or New Frontier?

- SWAT inherently implies quality/reliability relaxation aka approx. computing
- Key: Bound quality/reliability loss? (Subject to resource constraint)
  - Can serve as enabler for widespread adoption of resilience approach
  - Especially if automated
- Some other open questions
  - Instruction vs. data-centric?
  - Resiliency profiles and analysis at higher granularity?
  - How to compose?
  - Impact on recovery?
  - Other fault models?

Software doesn't ship with 100% test coverage, why should hardware?

Summary

- SWAT symptom detectors: <0.2% SDCs at near-zero cost
- Relyzer: Systematically find remaining SDC-causing app-sites
- Convert SDCs to detections with program-level detectors
- Tunable reliability vs. overhead
- Ongoing: Add tunable quality, Relyzer validations promising