Tolerating Silent Data Corruption (SDC) causing Hardware Faults through Software Techniques

IBM Research

AMD

Karthik Pattabiraman

Anna Thomas, Qining Lu, Jiesheng Wei, Bo Fang

University of British Columbia (UBC)

Meeta S. Gupta, Jude A. Rivers, IBM Research

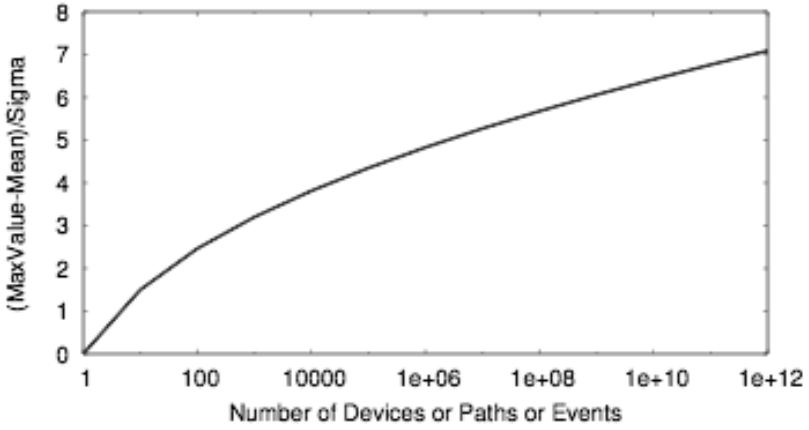
Sudhanva Gurumurthi, AMD Research

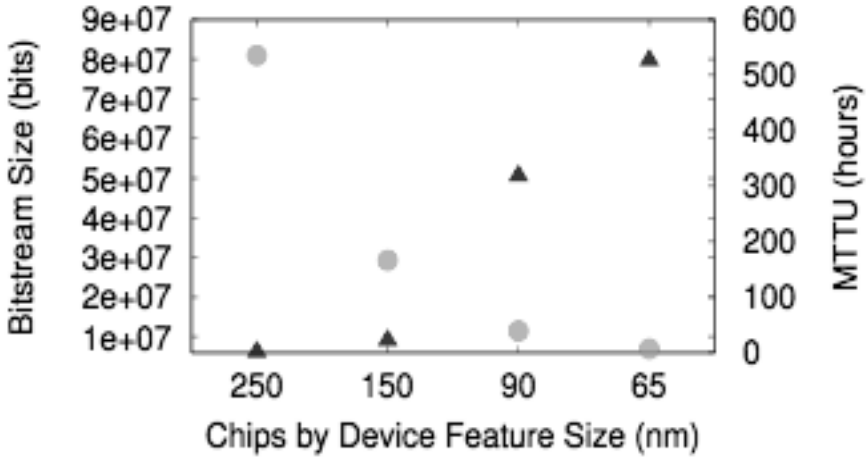
My Research

- **Building fault-tolerant and secure software systems**
- **Three areas**
  - Software resilience techniques [CASES'14][DSN'13][ISPASS'13]
  - Web applications' reliability [ICSE'14][ICSE'14][ESEM'13]
  - Smart meter security [HASE'14][WRAITS'12]
- **This talk: Software resilience techniques**

Motivation: Variations & Errors

- **Variation of device times**
  - Higher spread of device variations for future generations of technology
- **Feature size Vs MTTU**
  - Increase in number of bits correlated with decrease in MTTU of the chip



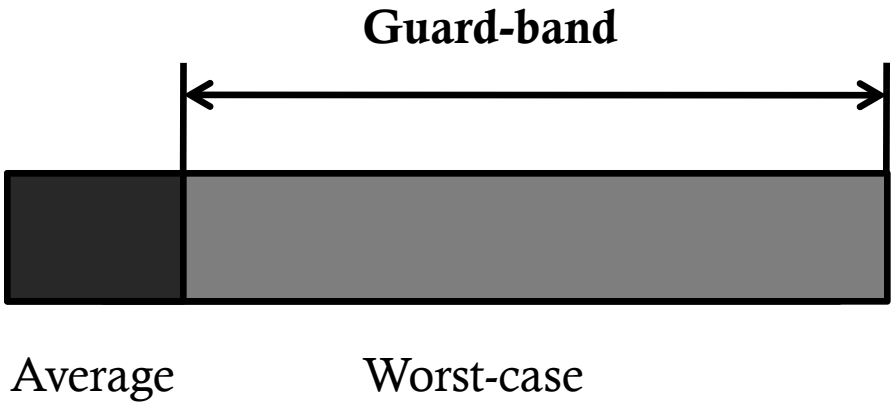


Source (CCC study on cross-layer reliability): www.relxlayer.org (2011)

Hardware Errors: "Solutions"

- **Guard-banding**

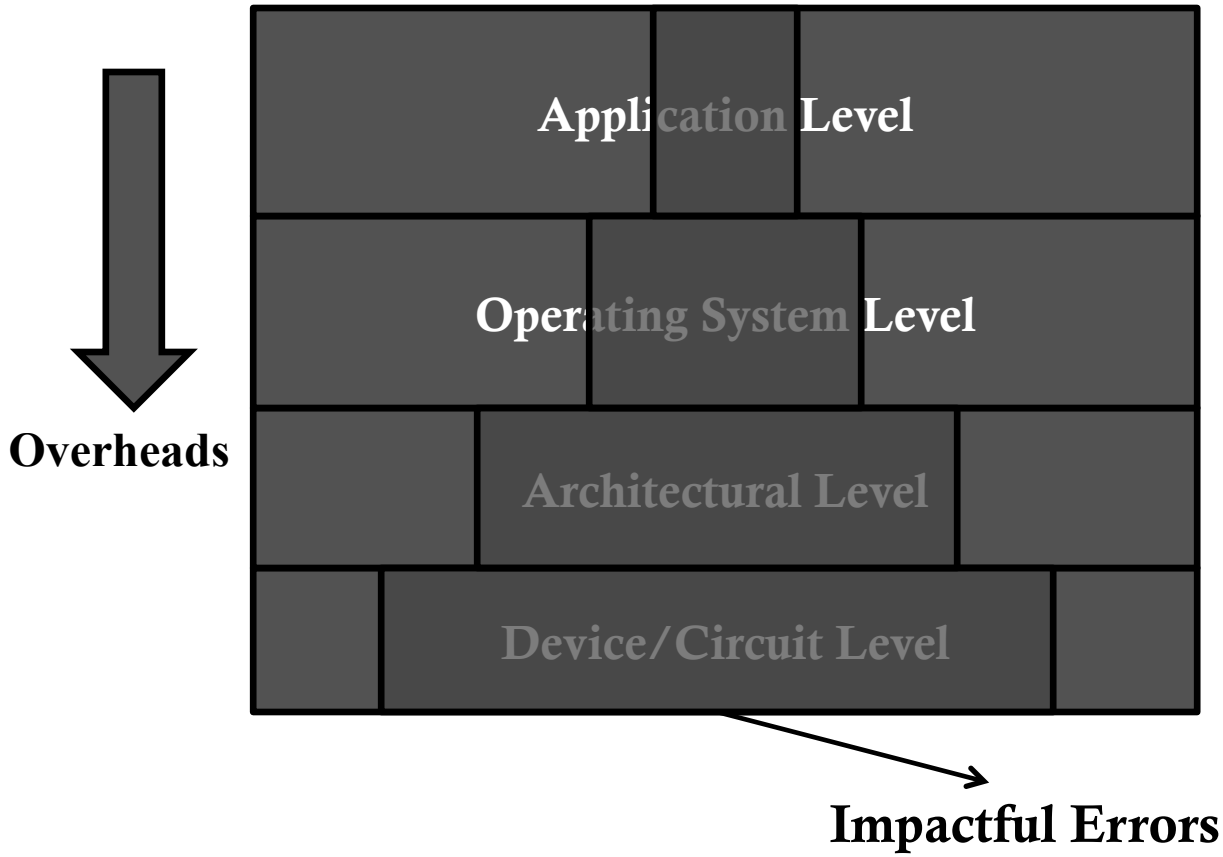
  Guard-banding wastes power and performance as gap between average and worst-case widens due to variations



- **Duplication**

  Hardware duplication (DMR) can result in 2X slowdown and/or energy consumption

Why Application-level techniques ?



Our Goal

- **Detect errors that cause Silent Data Corruption (SDC)**
  - Wrong results, Error Propagation etc.
- **Error Detection Coverage vs. Performance Overhead**
  - Achieve high SDC coverage while incurring low overhead by selectively protecting program instructions/data
- **No fault injections in applying to new programs**
  - Fault injections take significant time and effort

Outline

- Motivation and Goals
- **EDC Causing Error Detection [DSN'13][SELSE'13]**
- SDCTune: Protecting programs from SDCs [CASES'14]
- Error Resilience Characterization on GPUs [ISPASS'14]
- Conclusions and Future Work

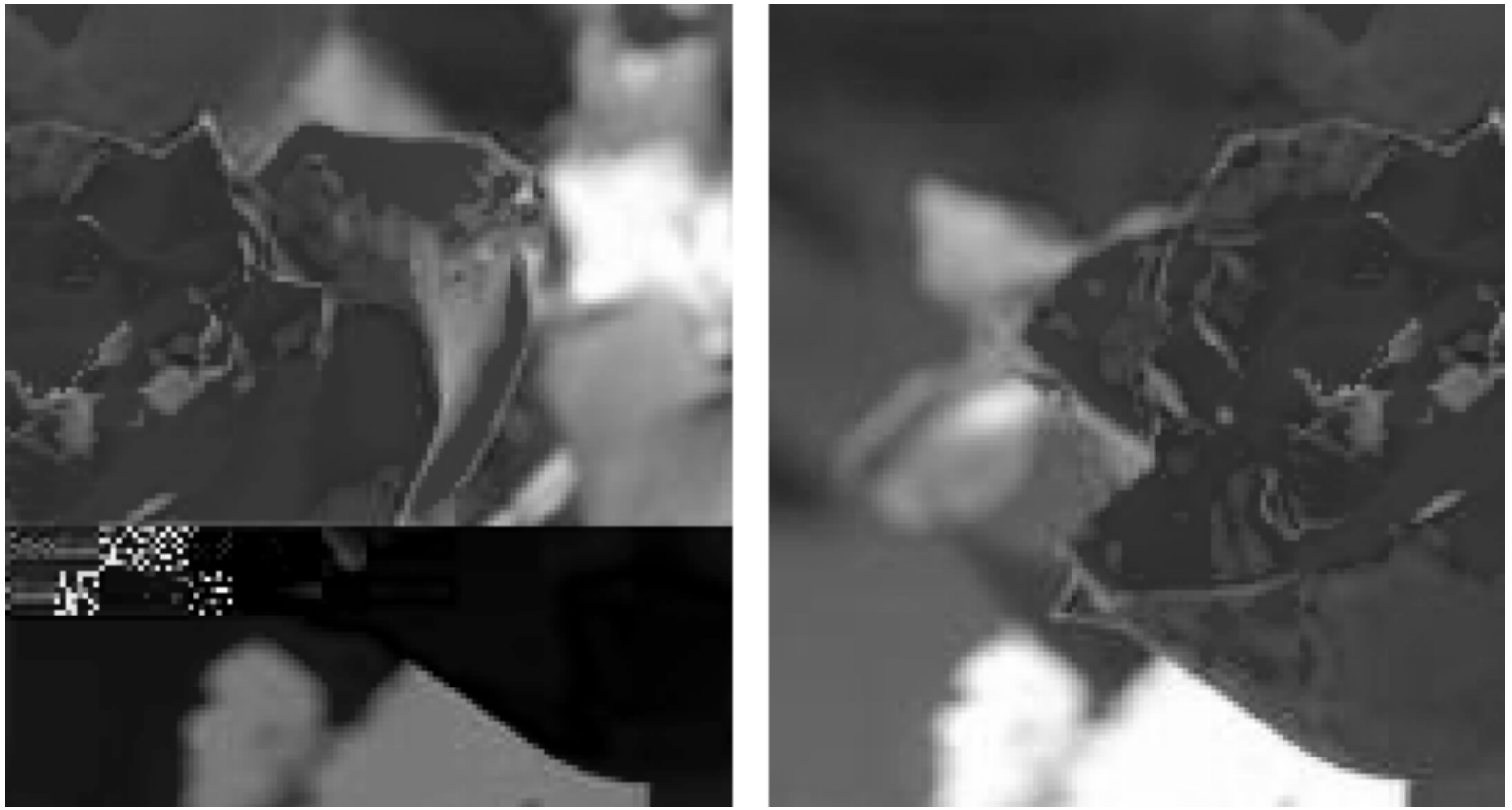
Soft Computing Applications

- Expected to dominate future workloads [Dubey'07]

Original image (left) versus faulty image from JPEG decoder

Egregious Data Corruptions (EDCs)

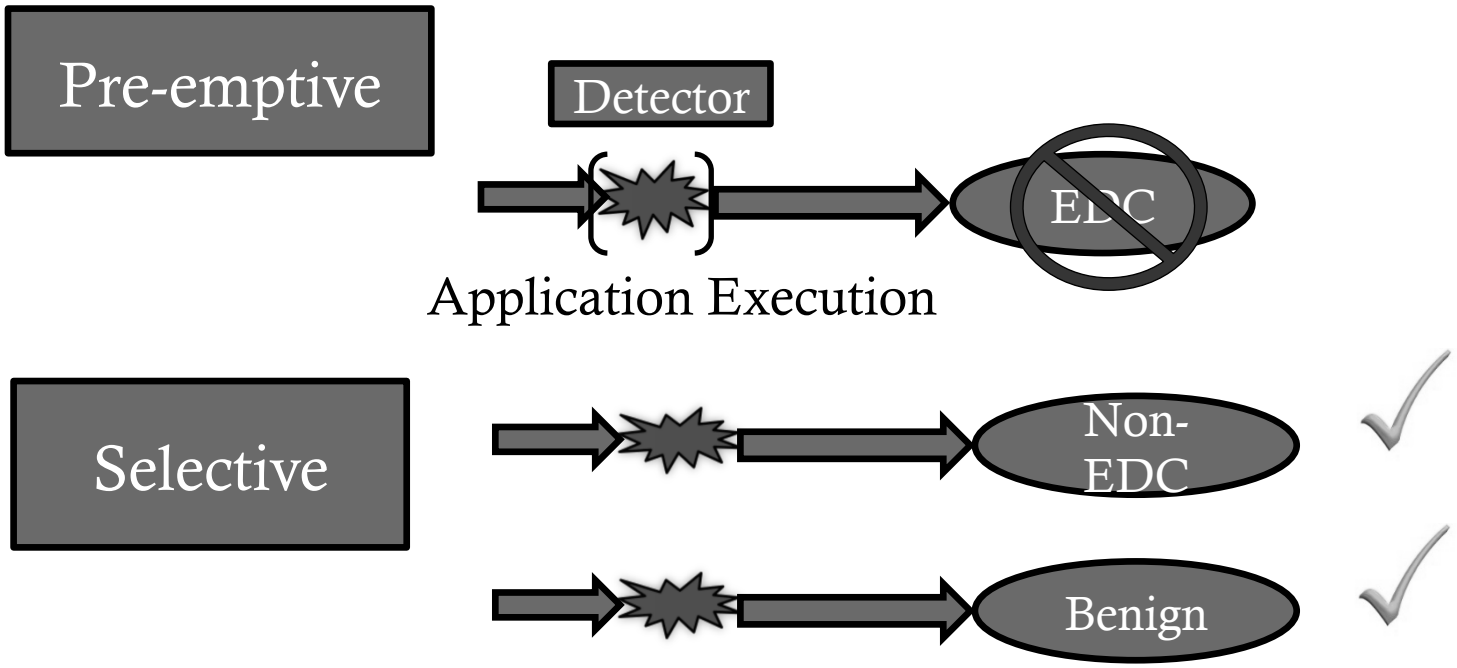
- Large or unacceptable deviation in output



EDC image (PSNR 11.37) vs Non-EDC image (PSNR 44.79)

Goal

- Detect EDC causing faults



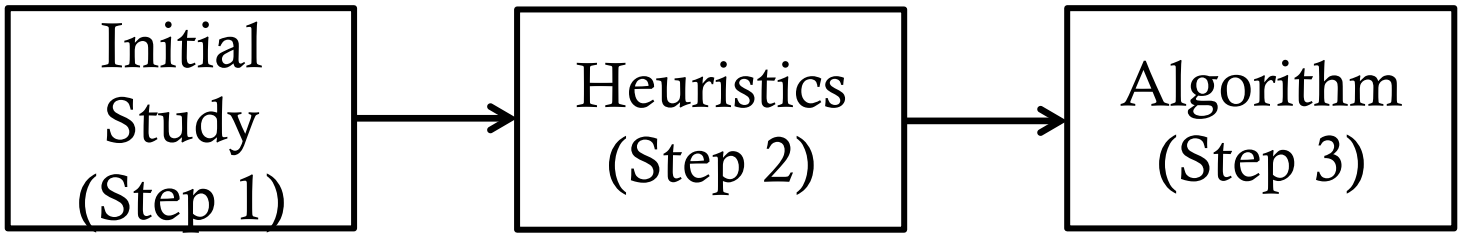
Fault model

- **Transient hardware faults**
  - Caused by particle strikes, temperature, etc.
  - Assume that program data is corrupted

- **Our Fault Model**
  - Single bit flip, One fault per run
  - Processor registers and execution units
  - Memory and cache protected with ECC

Approach

- Step 1: Separate EDCs from Non-EDCs by fault injections
- Step 2: Heuristics identifying code regions prone to EDC causing faults
- Step 3: Automated algorithm for detector placement

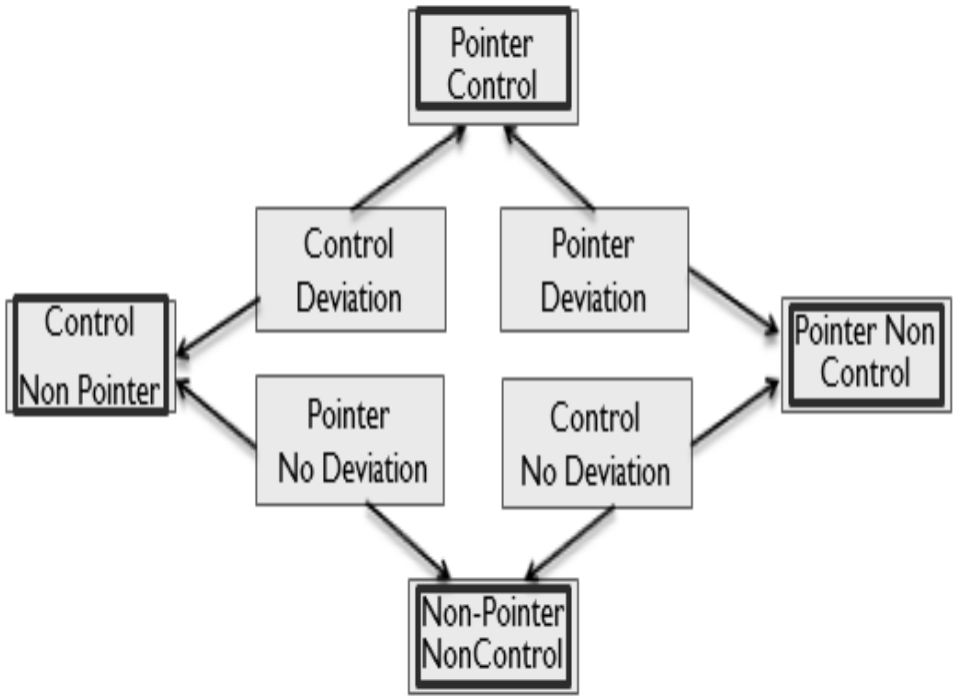


Step 1: Initial Study

- Correlation between data type – fault outcome

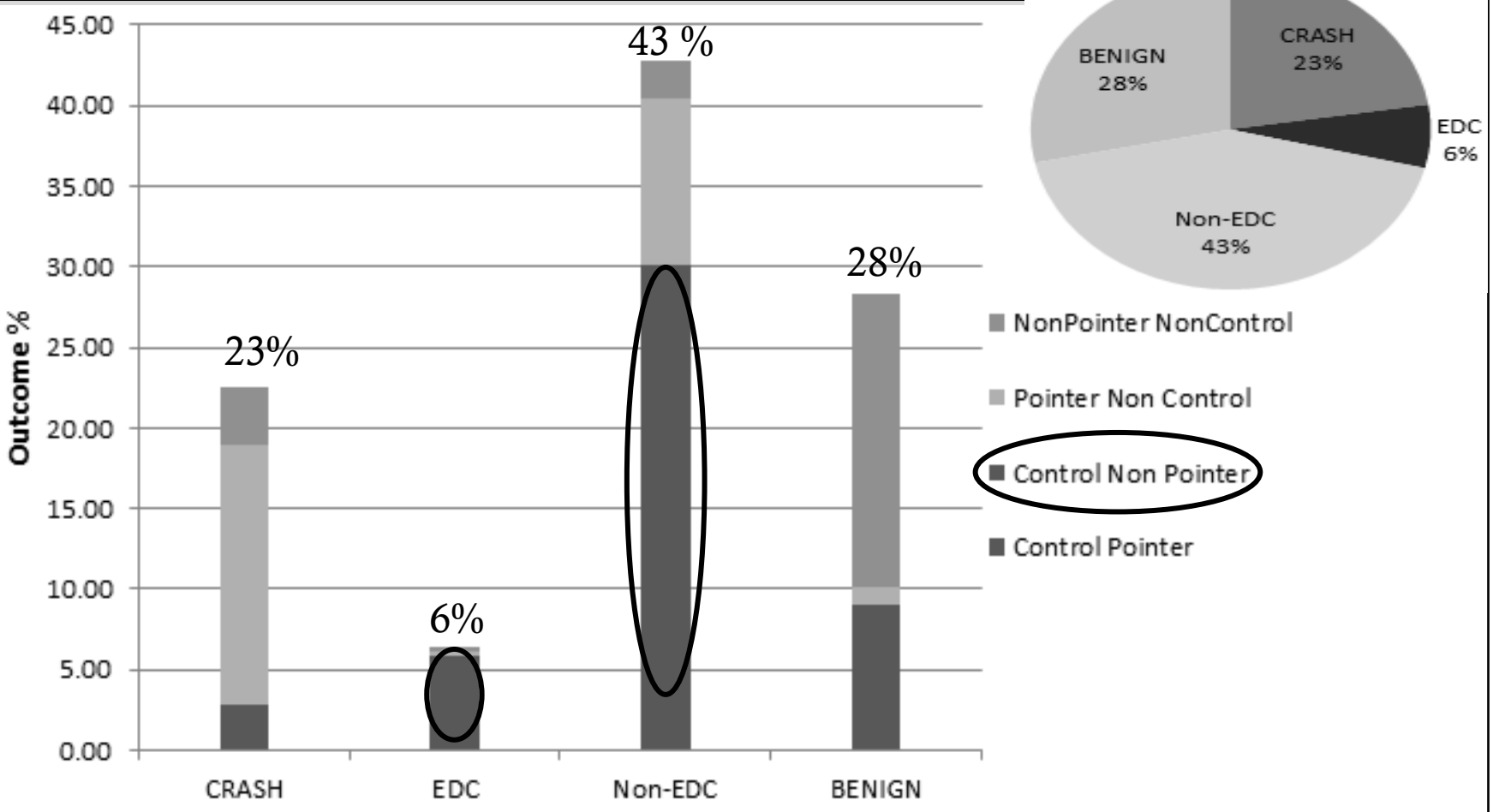
Monitor Control/Pointer Data

- Instrument code
- Fault Injection



Performed using LLFI [DSN'14]

Data Categorization: Faults



High correlation between Control Non-Pointer and EDC/Non-EDC

Step 2: Heuristics

```
void conv422to444 (char *src, char *dst,  int height, int width, int
offset) {
    for(j=0; j < height; j++){
      for(i=0; i < width; i++) {
        im1 = (i < 1) ? 0 : i – 1
        …
        dst[im1] = Clip[(21*src[im1])>>8];
      }
      if( j + 1 < offset) {
        src += w;
        dst += width; }
    }
}
```

Step 2: Heuristics

Faults affecting branches with large amount of data within branch body, has a higher likelihood of resulting in EDC outcomes

```
void conv422to444 (char *src, char *dst,  int height, int width, int
offset) {
    for(j=0; j < height; j++){
      for(i=0; i < width; i++) {
        im1 = (i < 1) ? 0 : i – 1
        …
        dst[im1] = Clip[(21*src[im1])>>8];
      }
      if( j + 1 < offset) {
        src += w;
        dst += width; }
    }
}
```

- Fault in offset
- Branch Flip

High EDC Likelihood

Step 2: Heuristics

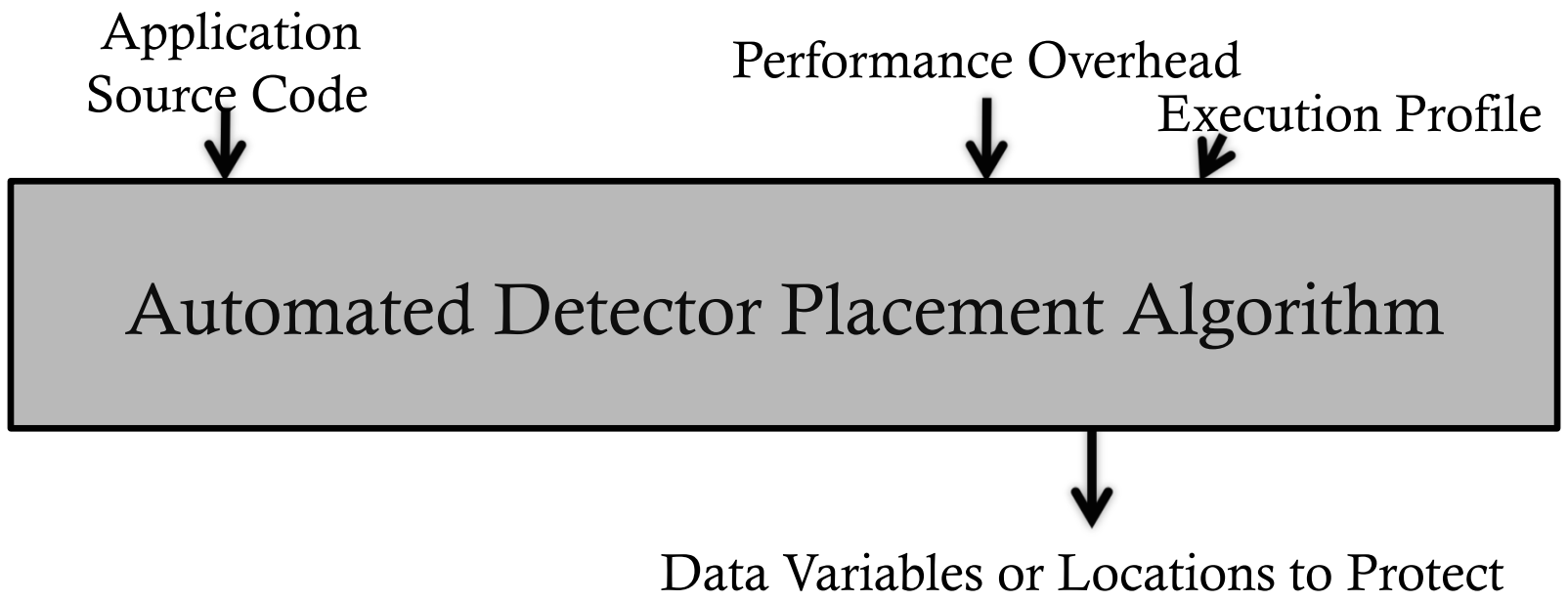
Faults affecting branches with large amount of data within branch body, has a higher likelihood of resulting in EDC outcomes

```
void conv422to444 (char *src, char *dst,  int height, int width, int
offset) {
    for(j=0; j < height; j++){
      for(i=0; i < width; i++) {
        im1 = (i < 1) ? 0 : i – 1
        …
        dst[im1] = Clip[(21*src[im1])>>8];
      }
      if( j + 1 < offset) {
        src += w;
        dst += width; }
    }
}
```

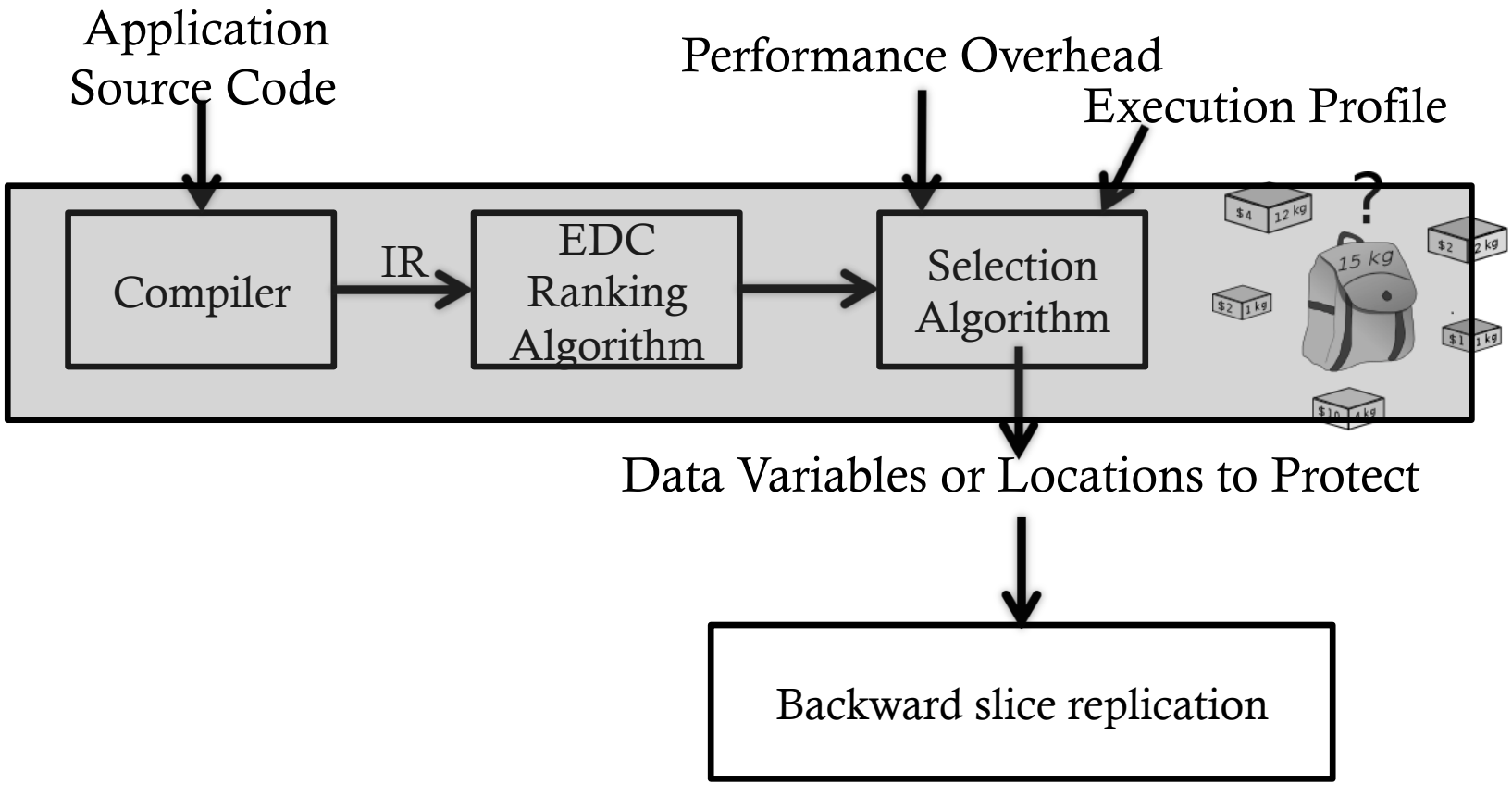
- Fault in result of branch

Low EDC Likelihood

Step 3: Algorithm



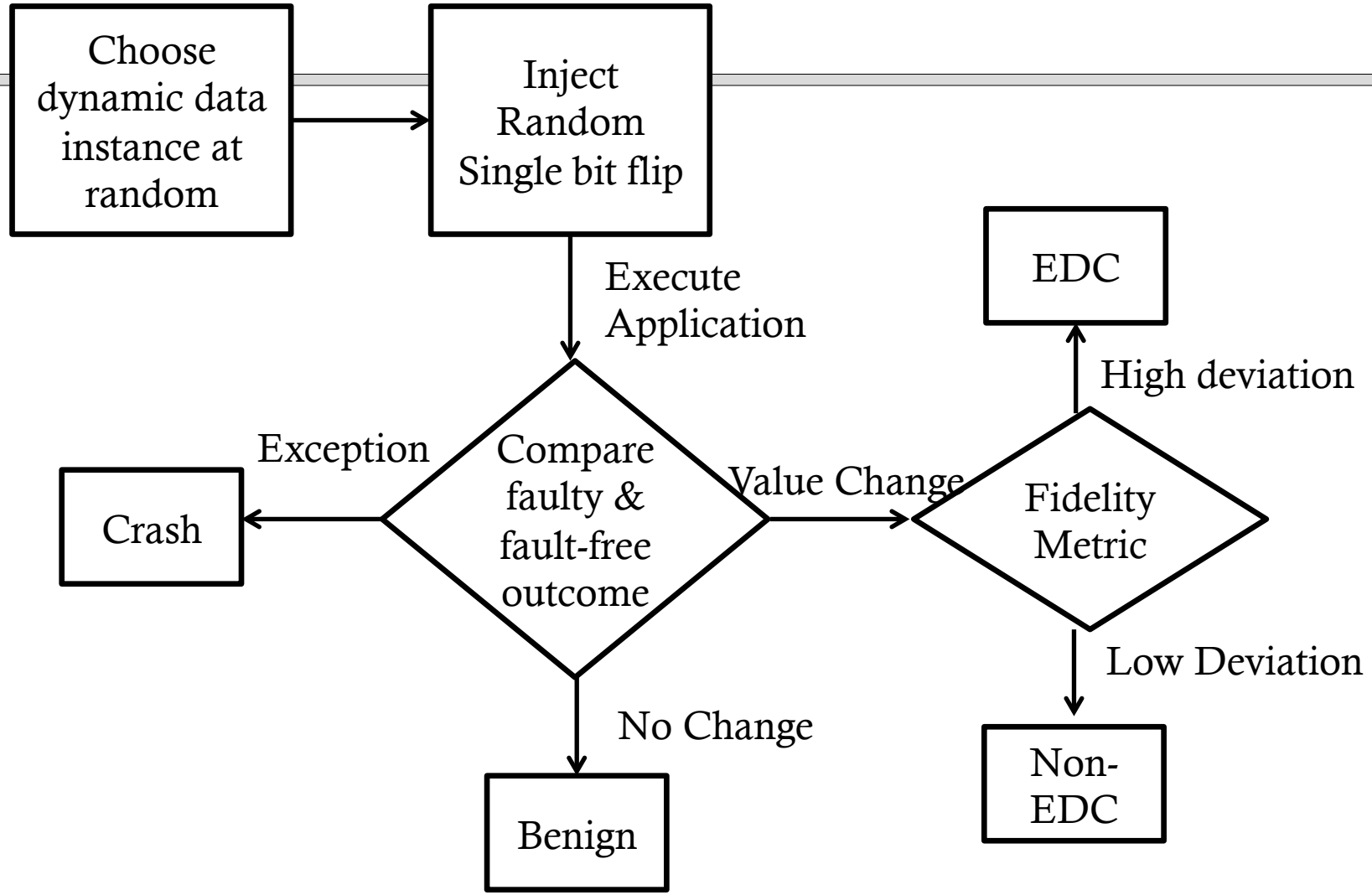
Step 3: Algorithm



Experimental Setup

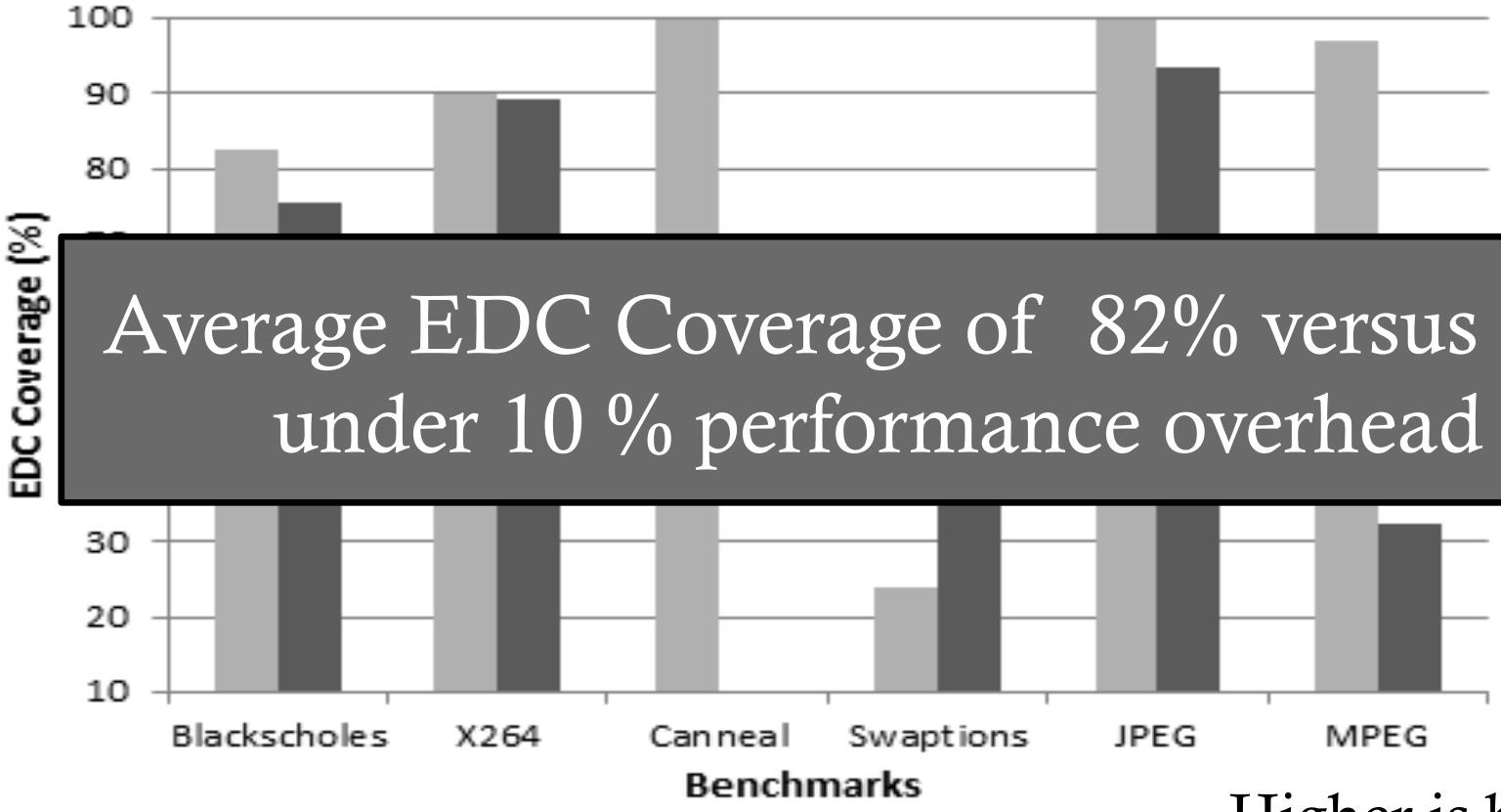
- Six Benchmarks from MediaBench, Parsec Suite
  - Fidelity Metric: PSNR, scaled distortion [Misailovic'12]
- Performed fault injections using LLFI [DSN'14]
  - 2000 fault injections, one fault per run (1.3% at 95% CI)
  - Validated with respect to assembly-level injectors for EDCs
- Measured EDC coverage under varying performance overhead bounds of 10, 20 and 25%

Experimental Framework



Coverage Evaluation

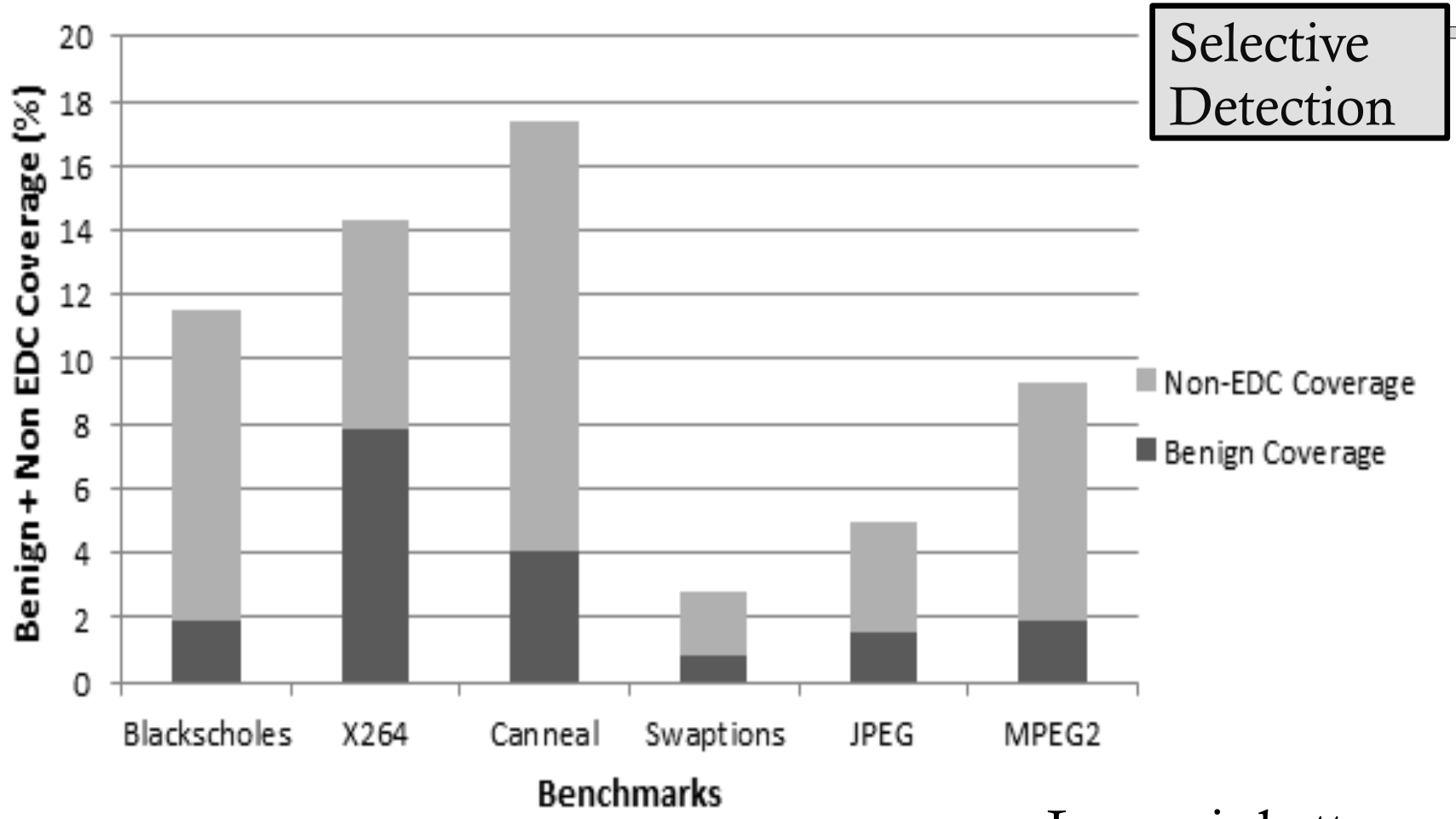
Pre-emptive Detection



Average EDC Coverage of 82% versus 56% under 10 % performance overhead

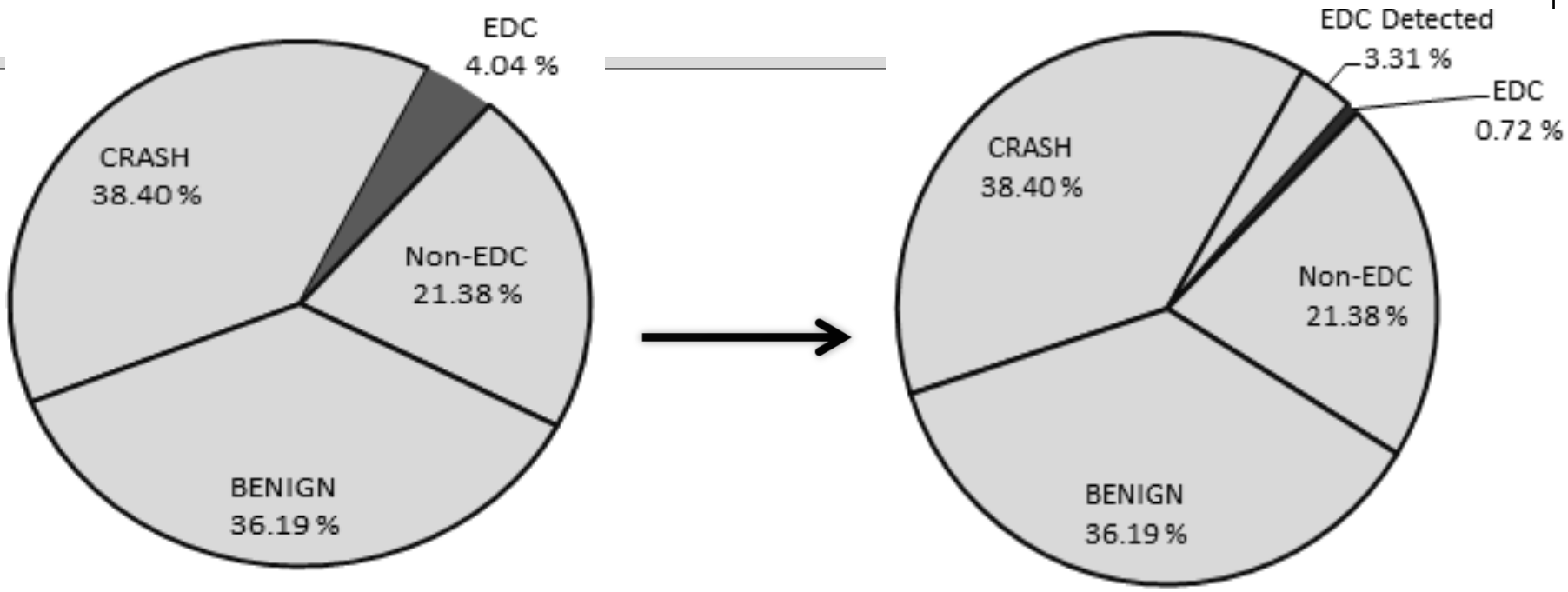
Higher is better

Coverage Evaluation



Lower is better

Coverage in Perspective



Our technique detects most EDC causing errors for a fraction of the cost of full duplication

Outline

- Motivation and Goals
- EDC Causing Error Detection [DSN'13][SELSE'13]
- **SDCTune: Protecting programs from SDCs [CASES'14]**
- Error Resilience Characterization on GPUs [ISPASS'14]
- Conclusions and Future Work

SDCTune: Goals

- **Earlier work on EDC causing errors showed feasibility of selectively protecting critical data**
  - Can we extend this to SDCs in general-purpose applications which are not as error resilient ?

- **Challenge:**
  - Not feasible to identify SDCs based on amount of data affected by the fault as was the case with EDCs
  - Need for comprehensive model for predicting SDCs based on static and dynamic program features

Main Idea

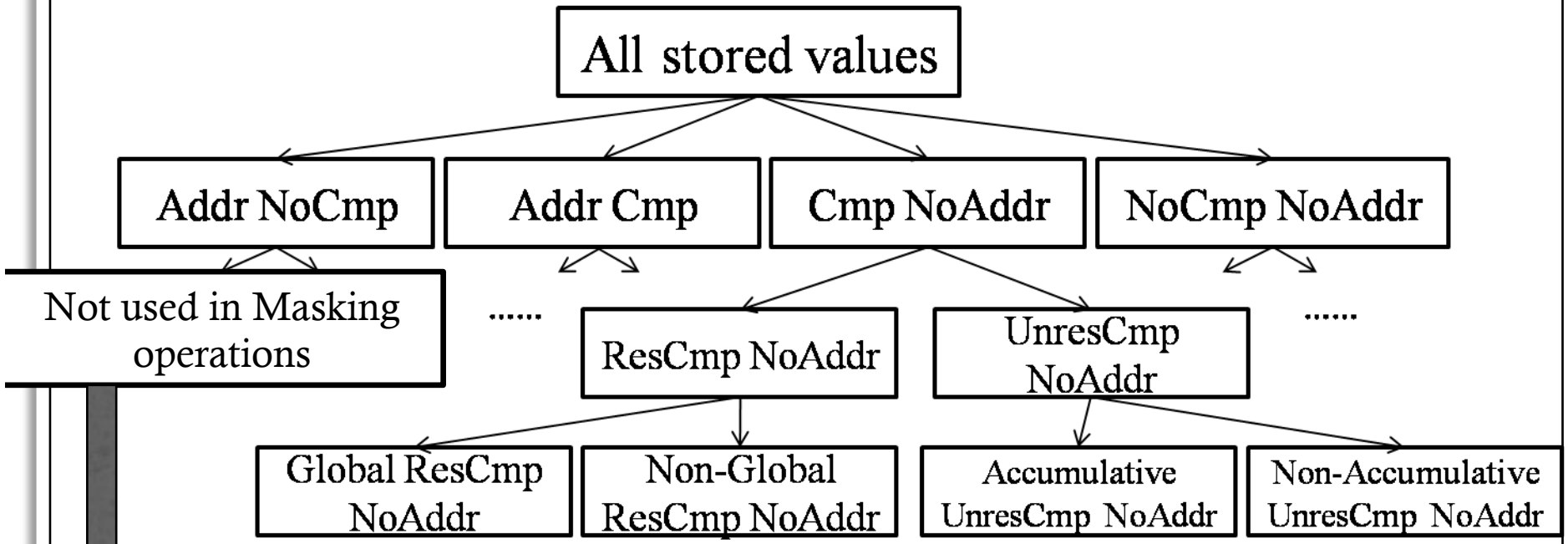
- **Start from "Store" and "Cmp" instructions propagate backward through data dependencies**
  - *"Store"* and *"Cmp"* are the end of visible data-dependency chain at the compiler levels

- **Predict P(SDC | Store or Cmp)**
  - *Extract* ***the related features*** *by static/dynamic analysis*
  - ***Quantify the effects*** *by classification and regression*
  - ***E*****stimate** *SDC rate of different "Store" and "Cmp" instructions*

Approach: Overview

- **Classification**
  - Classify the **stored values** and **comparison values** according to the extracted features (through static analysis)
  - Organize the features as a decision tree with each feature corresponding to a branch
- **Regression**
  - Within a single category, SDC rate may exhibit gradual correlations with several features
  - Use linear regression for the classified groups to estimate the SDC rate within a node of the tree

Approach: Decision Tree



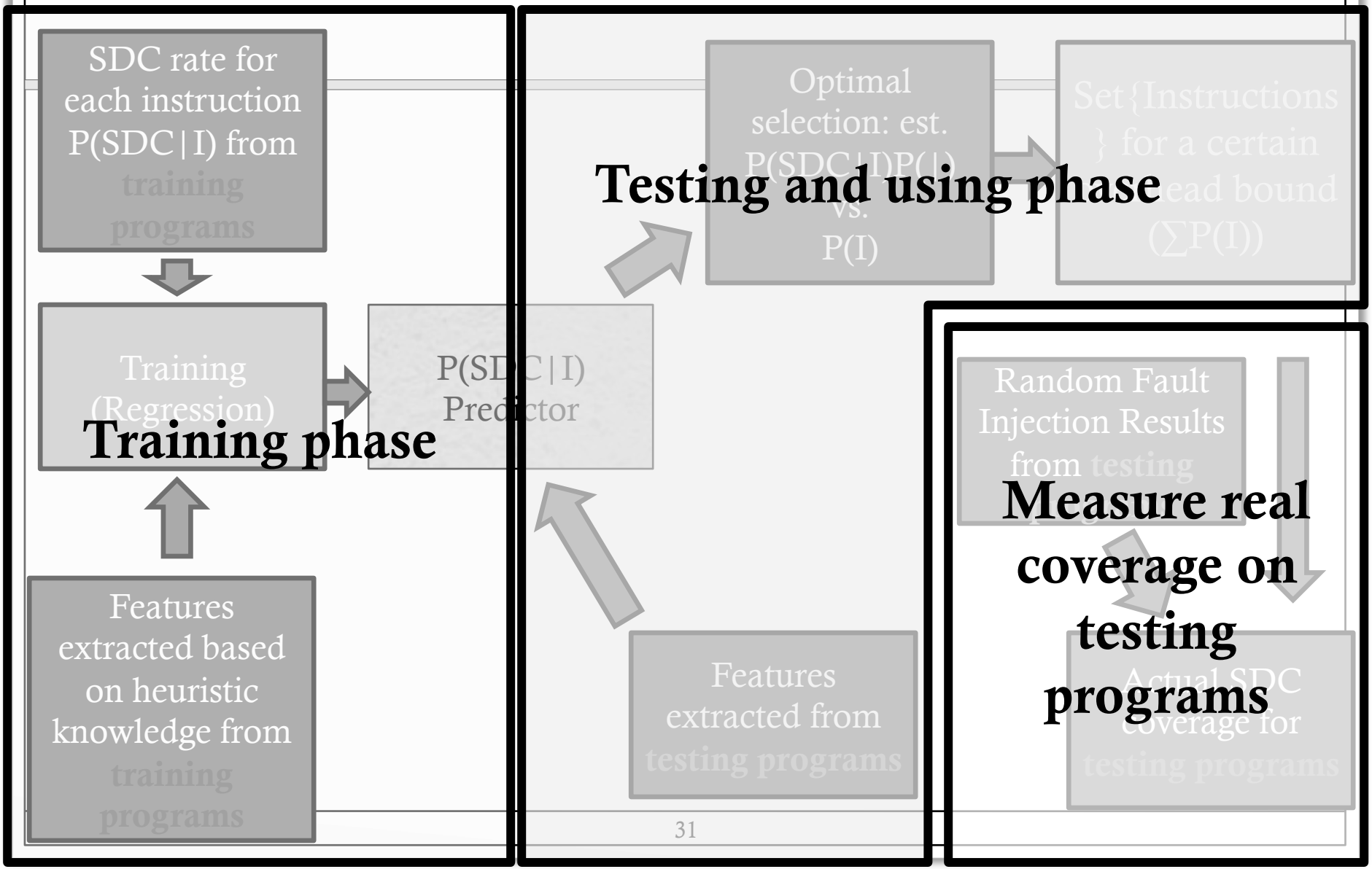
Example: ***Linear Regression for a Leaf***

$$PSDCI = -0.12 * data\ width + 0.878$$

Approach: Instruction Selection

- **Select instructions for Protected Set**
  - *Knapsack problem: value: estimated* ***P(SDC,I)****, weight:* ***P(I)***
  - ***A set of instructions to protect*** *for a given overhead bound*
  - Replicate static backward slices of the instructions to protect
- **Test coverage on training programs**
  - Measure the coverage for different overhead bounds and tune the model
  - Apply the model to a **different set of programs** to evaluate it

Experimental Methodology



Benchmarks

Training programs

Program	Description	Benchmark suite
IS	Integer sorting	NAS
LU	Linear algebra	SPLASH2
Bzip2	Compression	SPEC
Swaptions	Price portfolio of swaptions	PARSEC
Water	Molecular dynamics	SPLASH2
CG	Conjugate gradient method	NAS

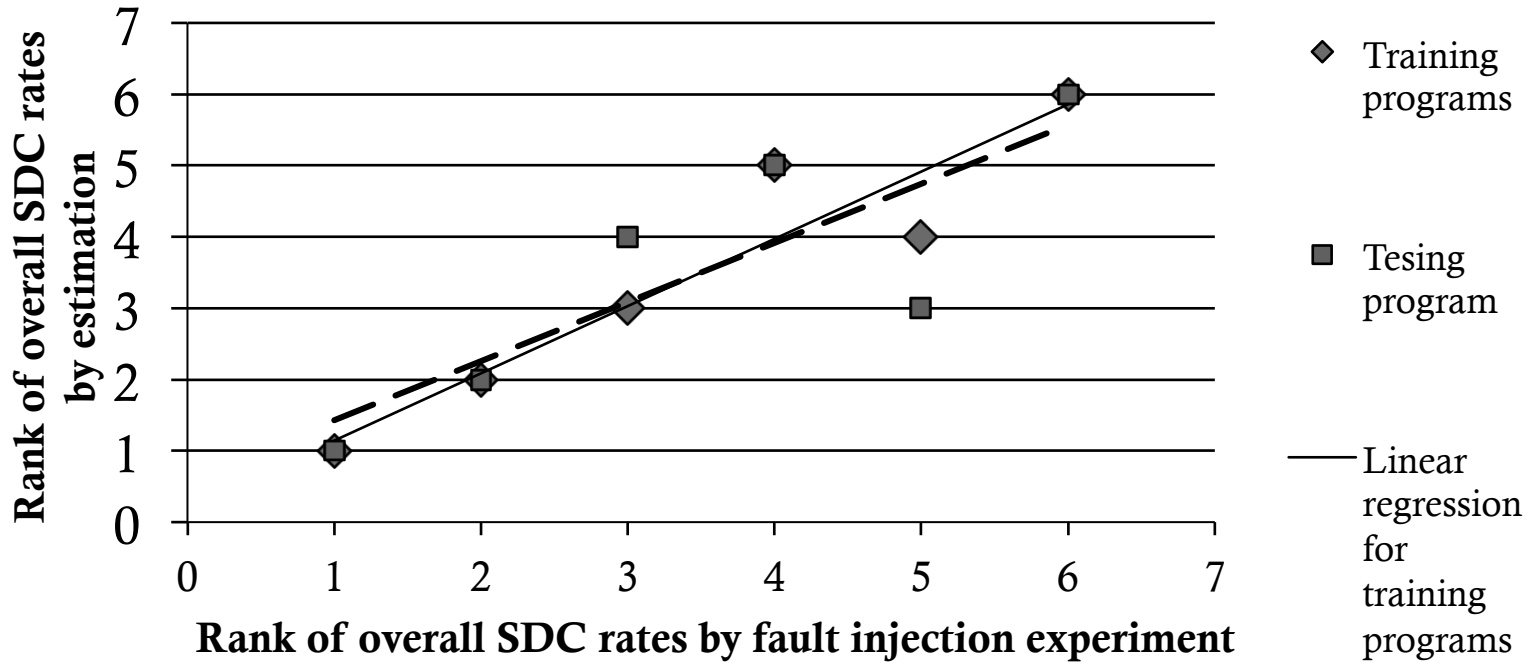
Testing programs

Program	Description	Benchmark suite
Lbm	Fluid dynamics	Parboil
Gzip	Compression	SPEC
Ocean	Large-scale ocean movements	SPLASH
Bfs	Breadth-First search	Parboil
Mcf	Combinatorial optimization	SPEC
Libquantum	Quantum computing	SPEC

Experiments

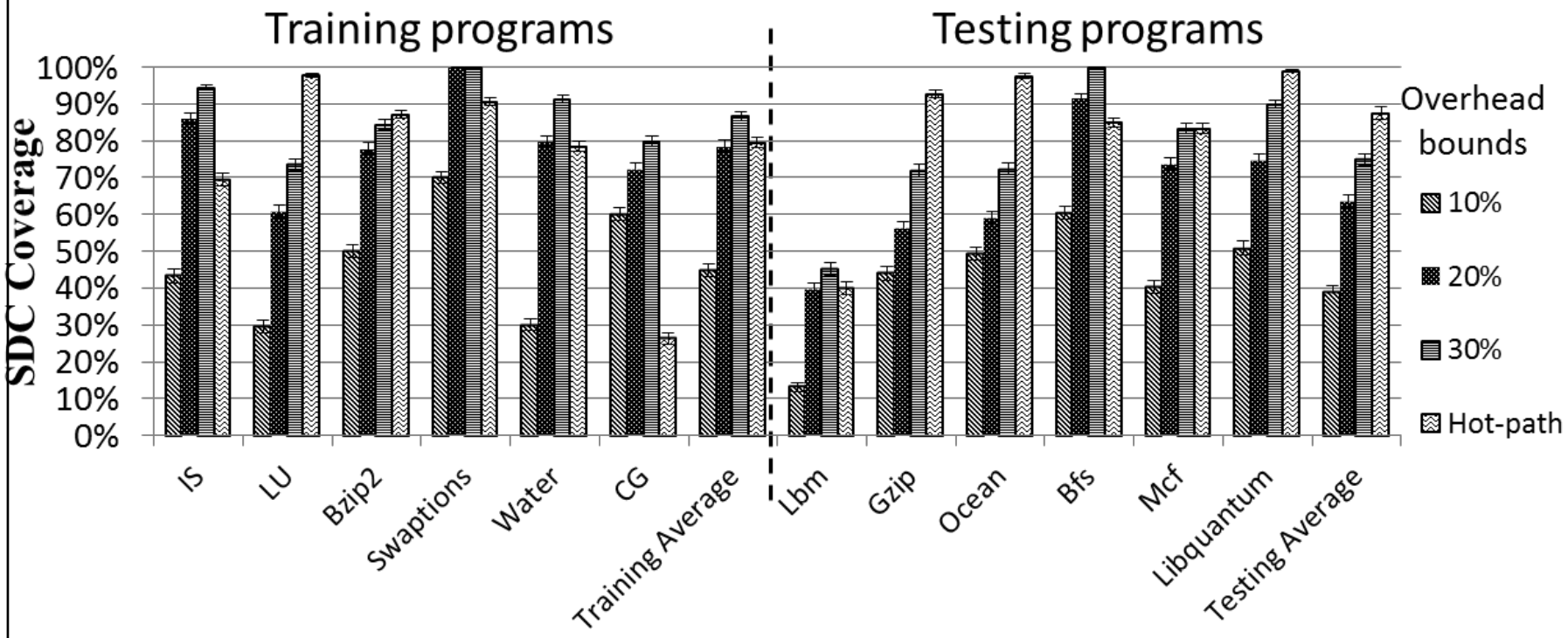
- Estimate overall SDC rates using SDCTune and compare with fault injection experiments
  - Measure correlation between predicted and actual
- Measure SDC Coverage of detectors inserted using SDCTune for different overhead bounds
  - Consider 10, 20 and 30% performance overheads
- Compared performance overhead and efficiency with full duplication and hot-path duplication
  - Efficiency = SDC coverage / Performance overhead

Overall SDC Rates: Ranks



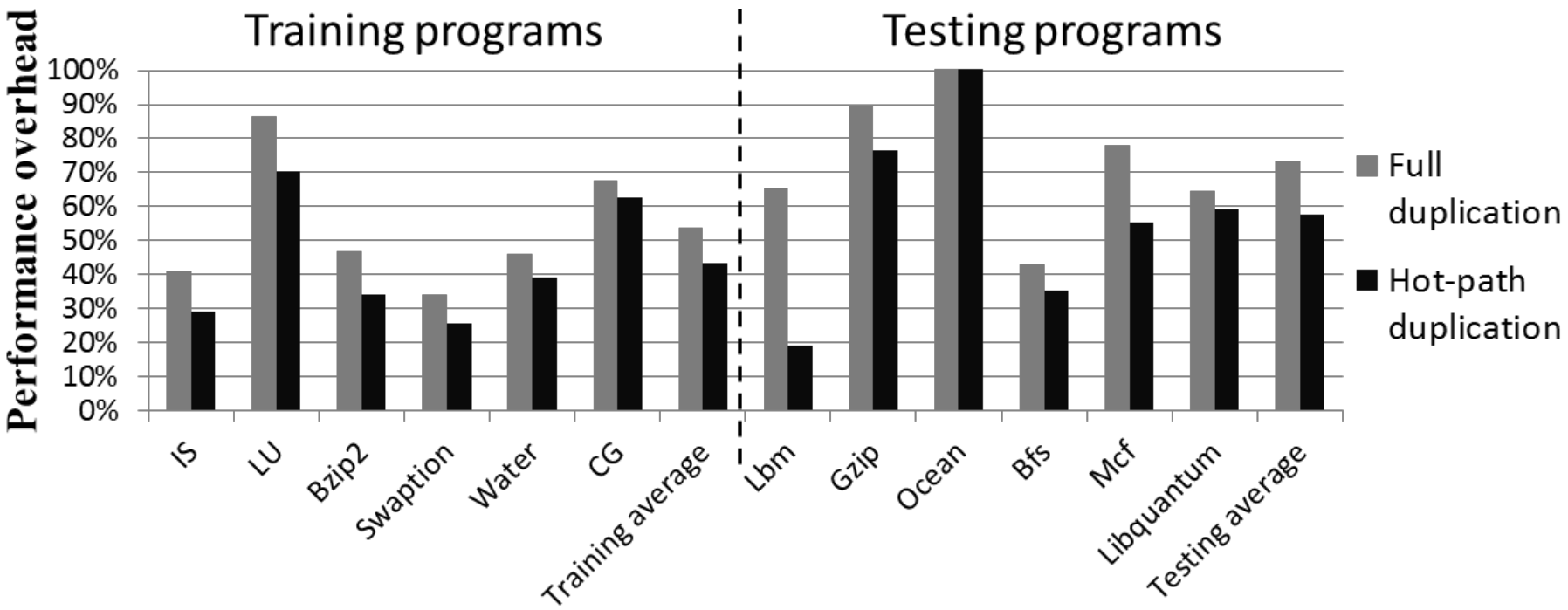
	Training programs	Testing programs
Rank correlation*	0.9714	0.8286
P-value**	0.00694	0.0125

SDC-Tune Based Detectors: SDC Coverage



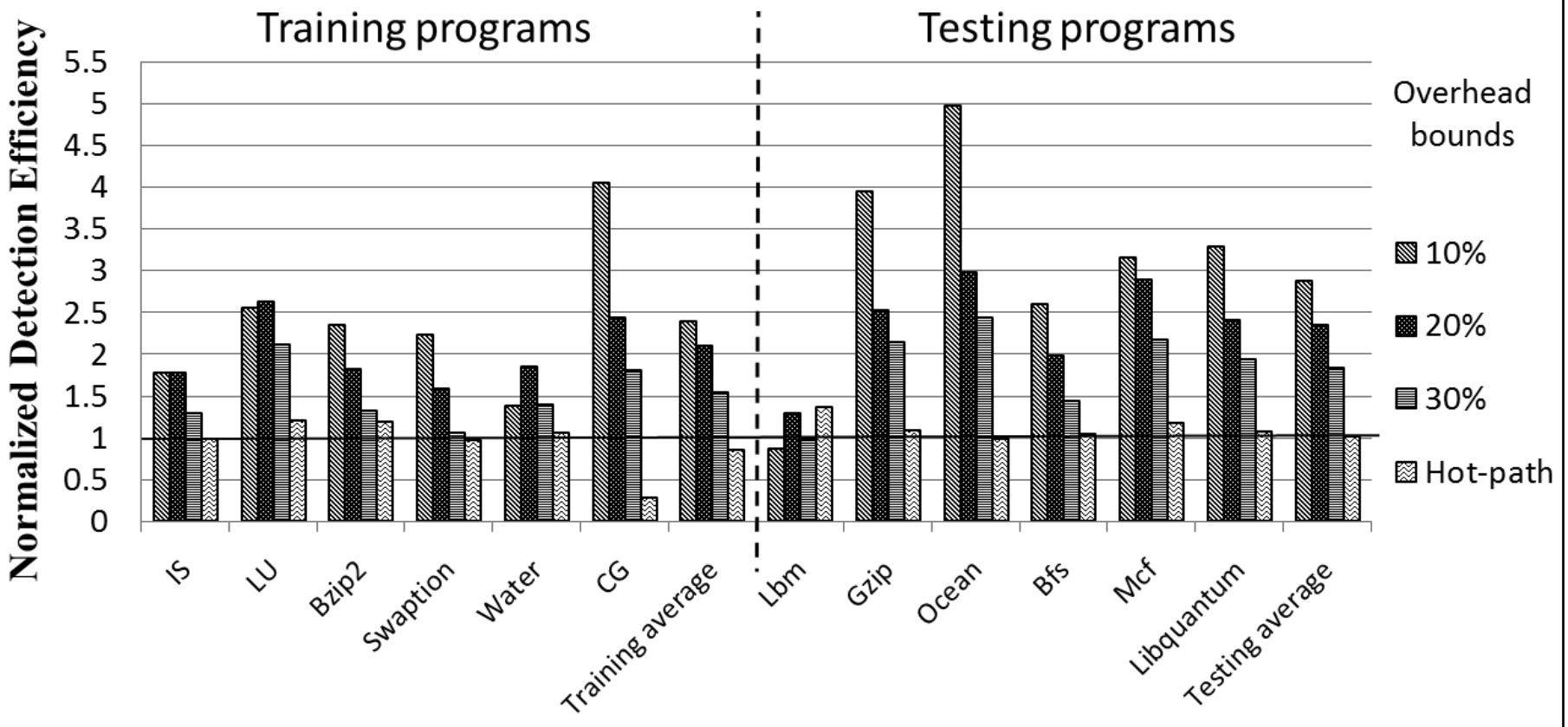
SDC coverage ranges from 45% to 87% for the training programs as overhead goes from 10 to 30%, while for testing programs it ranges from 39% to 75% for the same overhead.

Full Duplication and Hot Path Duplication: Overhead



Full duplication and hot-path duplication (top 10% of paths) have high overheads. For full duplication it ranges from 53.7% to 73.6%, for hot-path duplication it ranges from 43.5 to 57.6%.

Detection Efficiency of Detectors Vs. Full Duplication



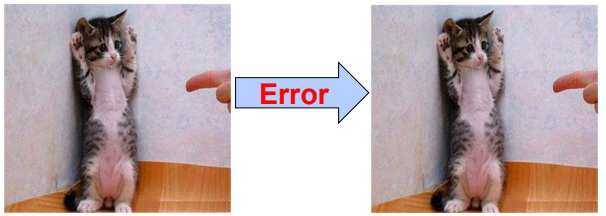
Detection efficiency of the detectors normalized to full duplication is 2.87x, 2.34x and 1.84x at the 10%, 20% and 30% overheads.

Outline

- Motivation and Goals
- EDC Causing Error Detection [DSN'13][SELSE'13]
- SDCTune: Protecting programs from SDCs [CASES'14]
- **Error Resilience Characterization on GPUs [ISPASS'14]**
- Conclusions and Future Work

GPU Error Resilience: Motivation

- GPUs have traditionally been used for error-resilient workloads
  - E.g. Image Processing



- GPUs are used in general-purpose applications, i.e. GPGPU
  - Small errors can lead to completely incorrect outputs

ATATTTTTTCTTGTT
TTTTATATCCACAAA
CTCTTTTCGTACTTT
TACACAGTATATCGT
GT

Error

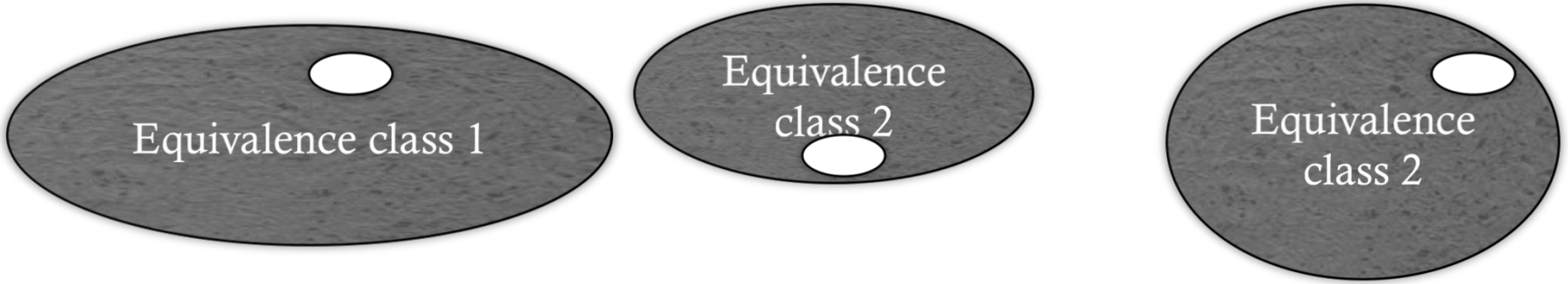
ATATTTTTTCTTGTT
TTTTATATCCACAAT
CTCTTTTCGTACTTT
TACACAGTATATCGT
GT

GPU Fault Injection: Challenges

- **Challenge 1: Scale of GPGPU applications**
  - GPGPU applications consist of thousands of threads, and injecting sufficient faults in each thread will be very time consuming
- **Challenge 2: Representativeness**
  - Need to execute application on real GPU to get hardware error detection
  - Need to uniformly sample the execution of the application to emulate randomly occurring faults

Addressing Challenge 1: Scale

- Choose representative threads to inject faults into
- Group threads with similar numbers of instructions into equivalence classes and sample from each class (or from the most popular thread classes)
- Hypothesis: Threads that execute similar numbers of instructions have similar behavior – validated by injection

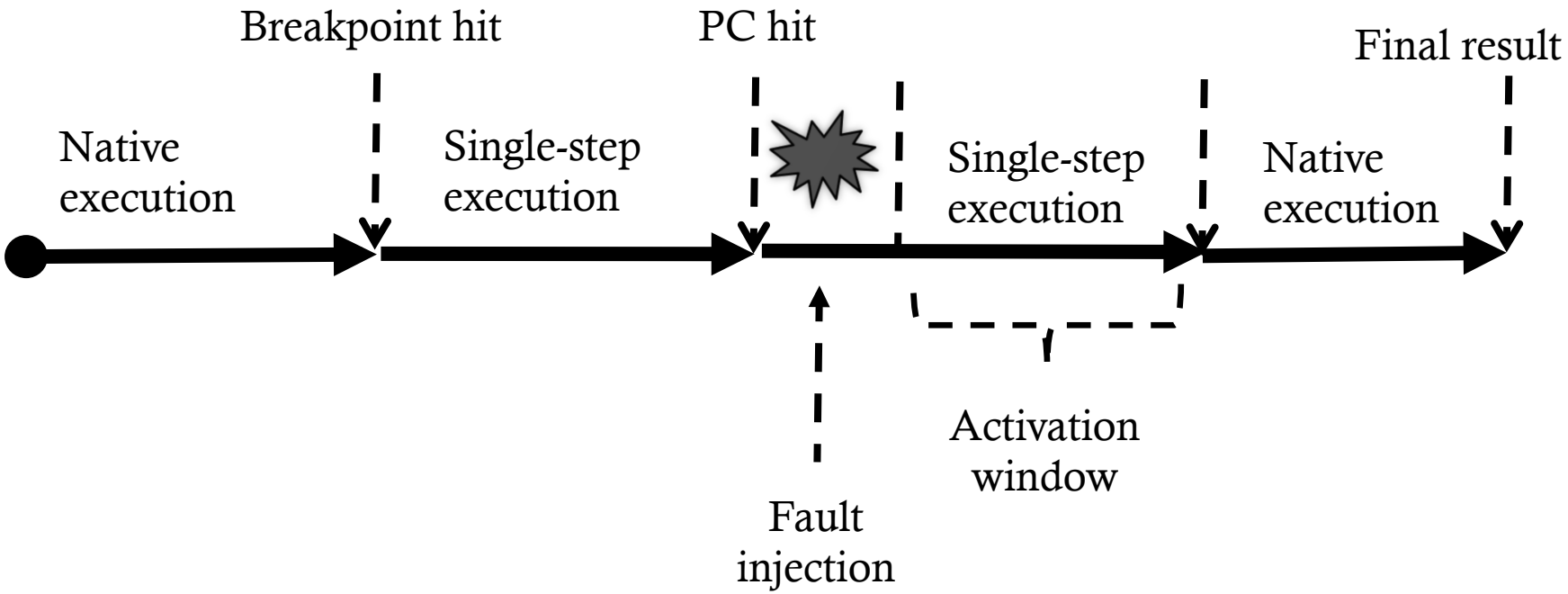


Addressing Challenge 2: Representativeness

- We use a source-level debugger for CUDA® GPGPU applications called CUDA-gdb
  - Advantage: Directly inject into the GPU hardware
  - Disadvantage: Requires source-code information to set breakpoints for injecting faults

- Our solution: Single-step the program using CUDA-gdb and map dynamic instructions to source code

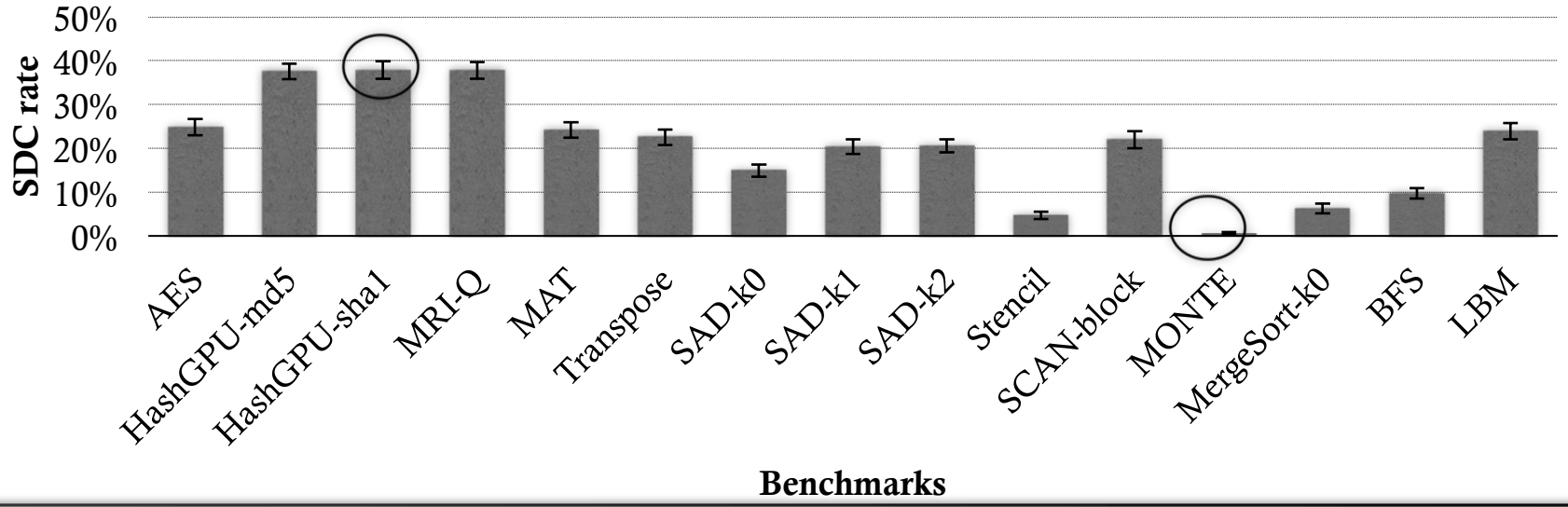
Fault injection Methodology: GPU-Qin



Experimental Setup

- NVIDIA® Tesla C 2070/2075
- 12 CUDA benchmarks comprising 15 kernels
  - Rodinia, Parboil and Cuda-SDK benchmark suites
- Only consider activated faults – faults read by application
- Outcomes
  - *Benign*: correct output
  - *Crash*: hardware exceptions raised by the system
  - *Silent Data Corruption (SDC)*: incorrect output, as obtained by comparing with golden run of the application
  - *Hang*: did not finish in considerably longer time

Overall Characterization Results



SDC Rates vary significantly across benchmarks (from 2 to 40%), which is much higher than in CPU applications (typically between 5 and 15%)

Hypothesis: Algorithmic Categories

Resilience Category	Benchmarks	Measured SDC	Dwarf(s) of parallelism
Search-based	MergeSort	6%	Backtrack and Branch+Bound
Bit-wise Operation	HashGPU, AES	25% - 37%	Combinational Logic
Average-out Effect	Stencil, MONTE	1% - 5%	Structured Grids, Monte Carlo
Graph Processing	BFS	10%	Graph Traversal
Linear Algebra	Transpose, MAT, MRI-Q, SCAN-block, LBM, SAD	15% - 25%	Dense Linear Algebra, Sparse Linear Algebra, Structured Grids

Implications of our Results

- Wide variation in SDC rates across GPGPU applications, much more than CPU applications
  - Need for application specific fault-tolerance
- Correlation between algorithm and error resilience
  - Can be used to obtain quick estimates without FI
  - Can be used to customize level of protection provided

Outline

- Motivation and Goals
- EDC Causing Error Detection [DSN'13][SELSE'13]
- SDCTune: Protecting programs from SDCs [CASES'14]
- Error Resilience Characterization on GPUs [ISPASS'14]
- **Conclusions and Future Work**

Conclusion

- Selective protection of instructions in applications for both detecting both EDCs and SDCs
  - Protection configurable based on max performance overhead
  - Can provide high detection coverage for most severe errors
- GPGPU applications have wider variations in SDC rates compared to CPU applications
  - Correlation between algorithmic properties and application error resilience → mapping to dwarves
  - Development of new algorithms for resilient computation

Fault Injectors at http://github.com/DependableSystemsLab

Future Work

- Understanding effect of algorithm on shared memory parallel applications on the CPU
  - Similar correlations as GPGPU apps [FTXS'14]
- Effect of compiler optimizations on the error resilience of applications [AER'13]
  - Identify safe optimizations for given error resilience targets
- Theoretical foundations of programs' error resilience
  - PVF analysis combined with heuristics-based analysis