DATE 14
Design, Automation & Test in Europe
24-28 March, 2014 - Dresden, Germany
The European Event for Electronic System Design & Test

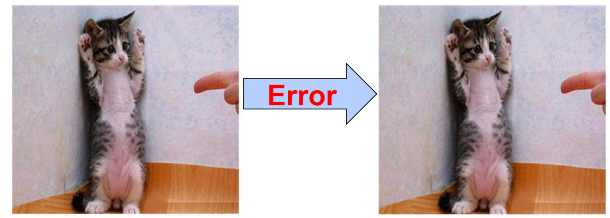
Evaluating the Robustness of GPU Applications through Fault Injection

Karthik Pattabiraman, Bo Fang, Matei Ripeanu,
University of British Columbia (UBC)

in collaboration with
Sudhanva Gurumurthi (AMD Research)

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

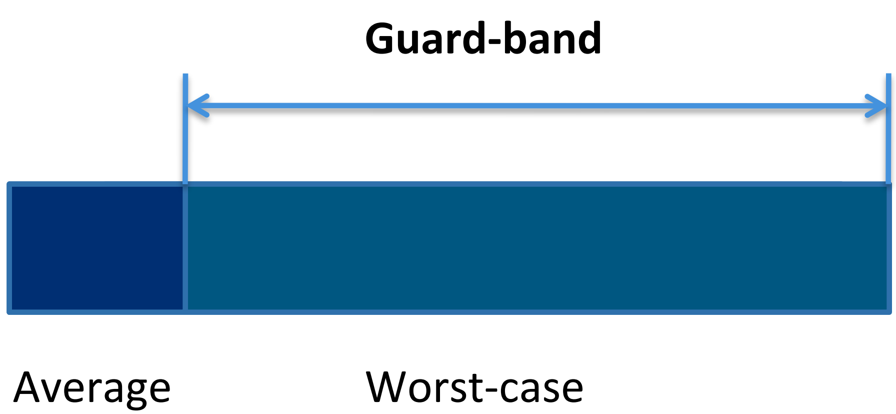
Error

ATATTTTTTCTTGTT
TTTTATATCCACAAT
CTCTTTTCGTACTTT
TACACAGTATATCGT
GT

Hardware Errors: Hardware Solutions

- **Guard-banding**

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

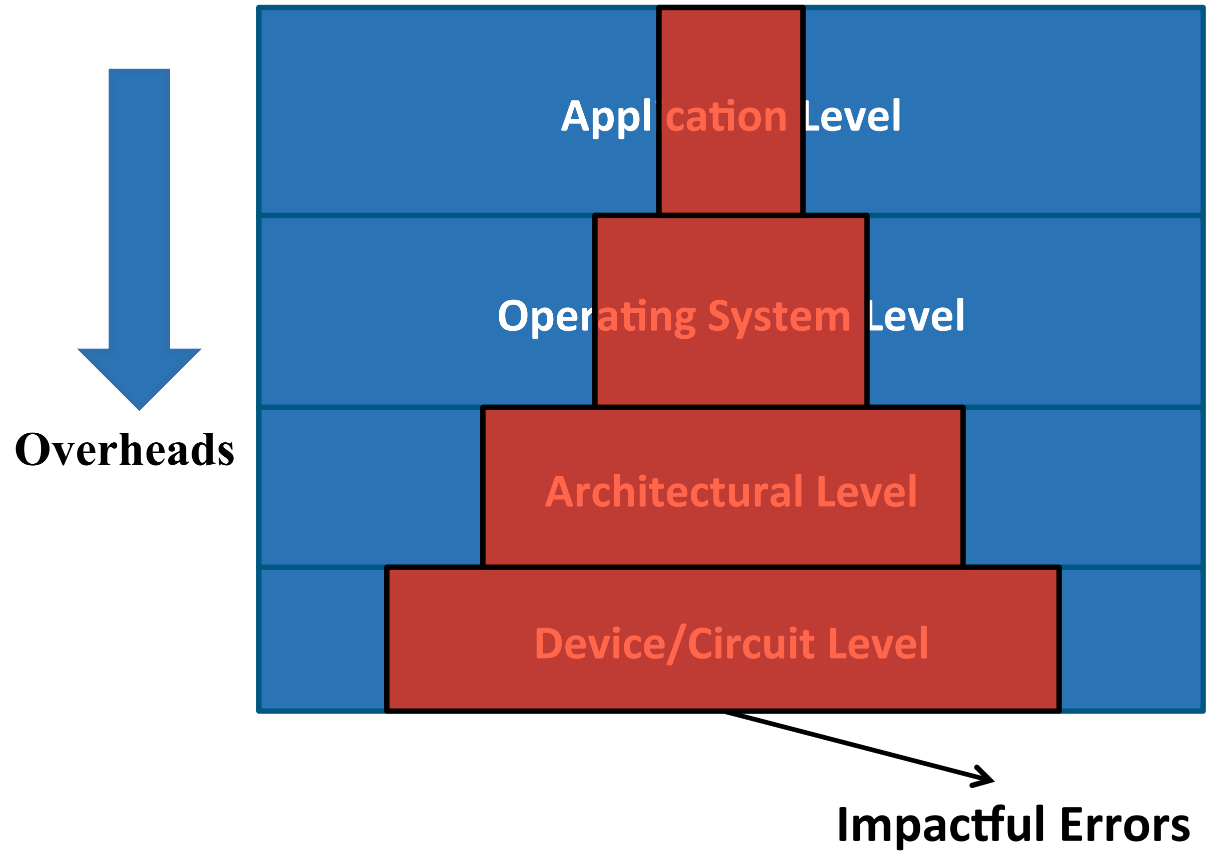


- **Duplication**

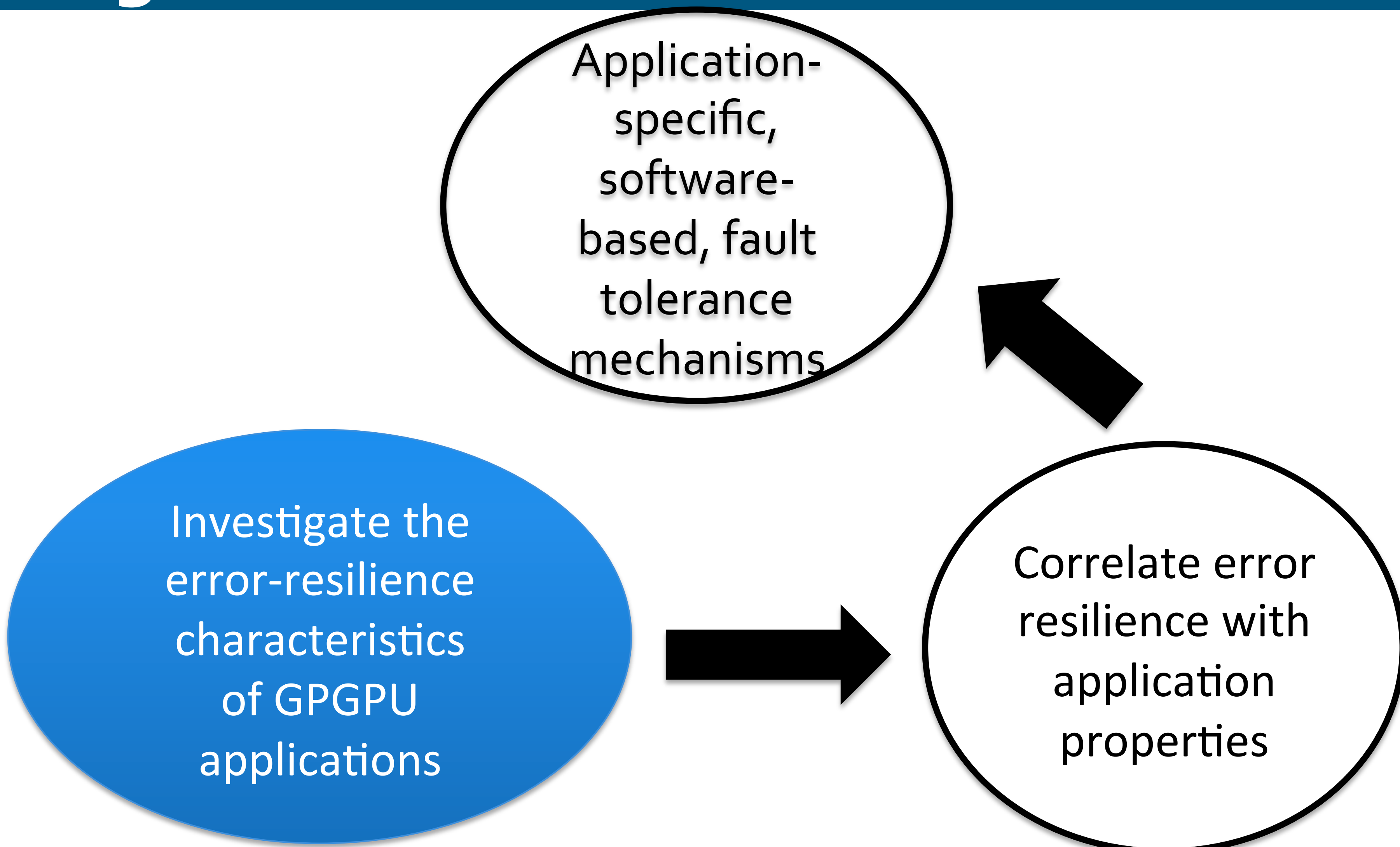
  Hardware duplication (DMR) can result in 2X slowdown and/or energy consumption

Why Software Solutions?

Errors get progressively filtered as we go up the system stack

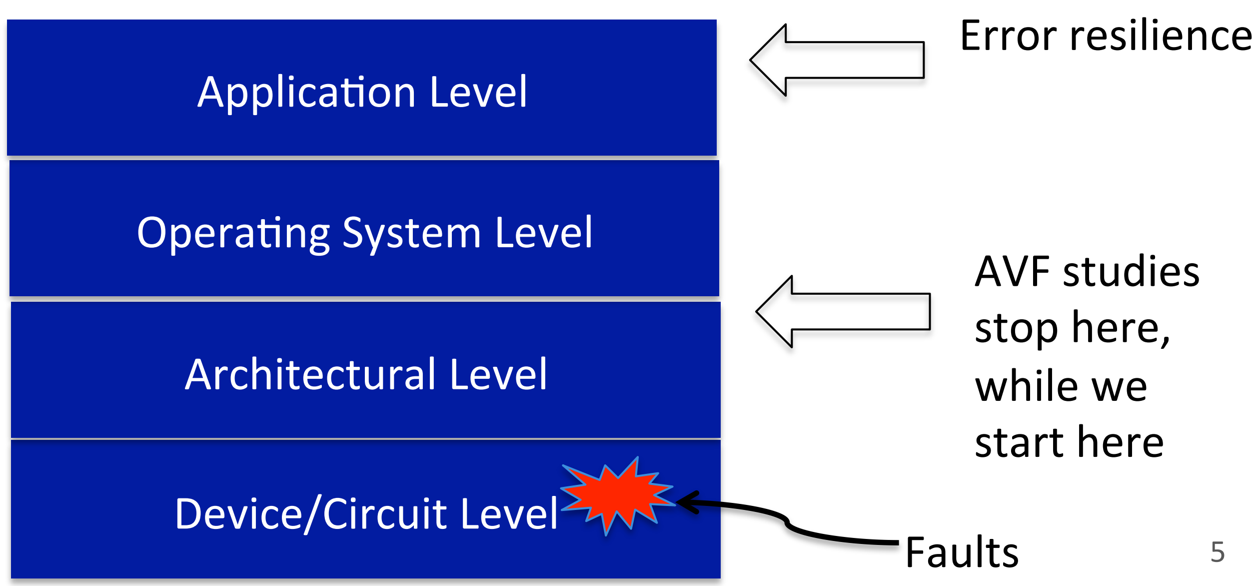


High-level Goal



Vulnerability Vs. Resilience

- **Vulnerability:** *Probability that the system experiences a fault that causes a failure*
  - Do not consider the behavior of applications
- **Error Resilience:** *Given a fault in the application, what is the probability that the application completes correctly ?*



Fault Model

- **Transient faults (e.g., soft errors): Single bit-flips**

- **Faults in**
  - Arithmetic and Logic Unit (ALU)
  - Floating Point Unit (FPU)
  - Load-Store Unit (LSU)

- **Do not consider faults in memory elements or registers**
  - Assumes ECC protection (e.g., NVIDIA Fermi)

Software Fault Injection (SWiFI)

- **Perturb the application state to emulate the effects of errors, and measure its resilience to the errors**
  – Execute the application **to completion** under the error to study the end-to-end effects of errors
  – Studies the actual effects of the fault instead of estimating the worst-case probabilities like AVF

- **Many fault injectors for CPUs**
  – NFTAPE, Goofi, Xception, FERRARI
  – **No fault injector for GPUs**

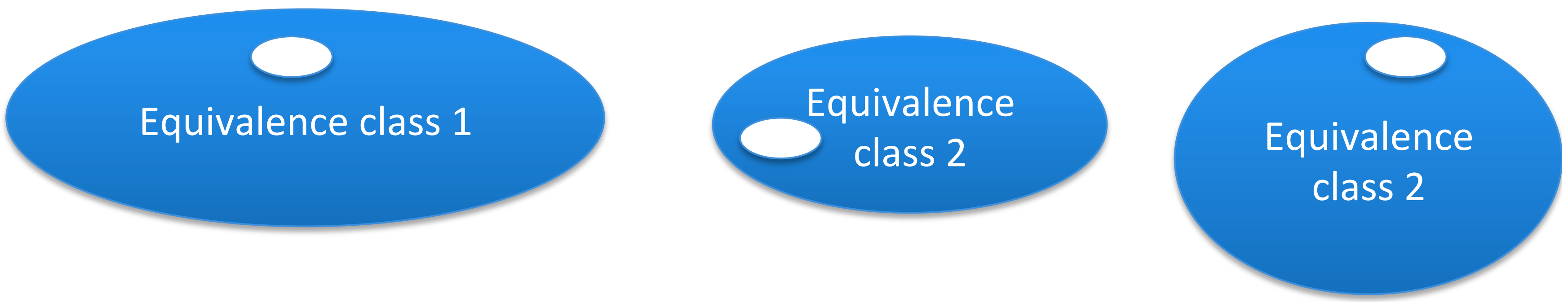
GPU Fault Injection: Challenges

- **Challenge 1: Scale of GPGPU applications**
  - GPGPU applications consist of hundreds 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



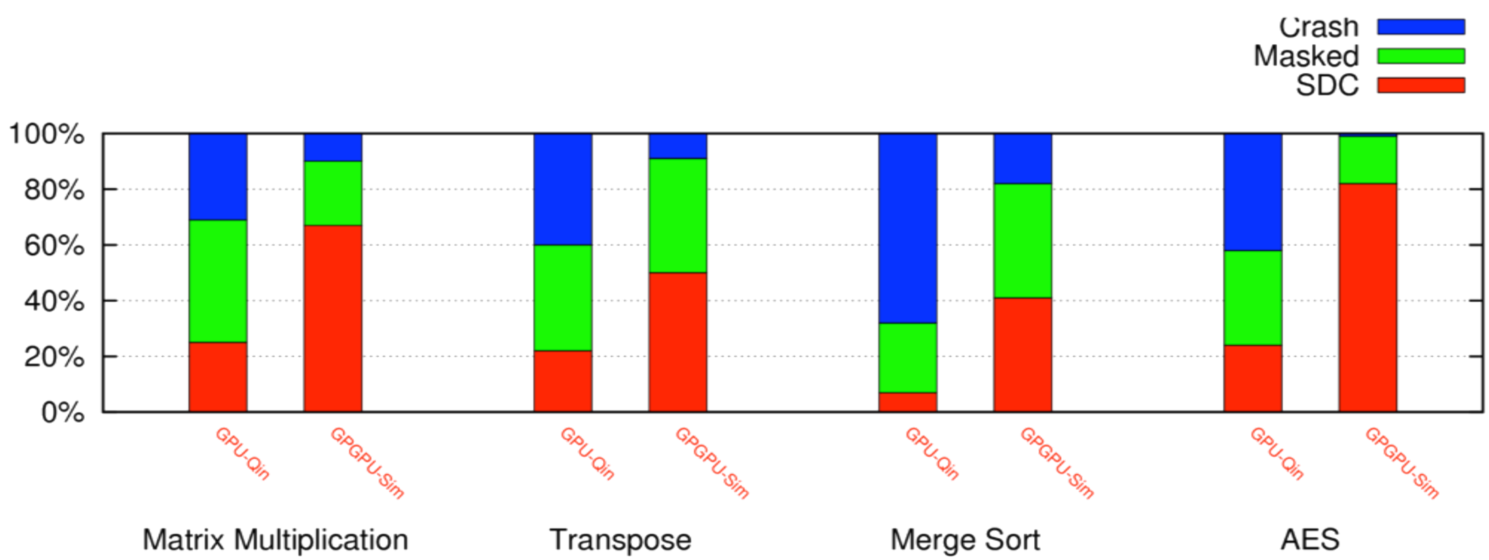
Addressing Challenge 1: Scale

Most applications have a single class or less than 5 equivalence classes -> Random sampling covers > 95% of the threads in these applications (Exception is BFS).

Category	Benchmarks	Groups	Groups to profile	% of threads in picked groups
Category I	AES,MRI-Q,MAT,Mergesort-k0, Transpose	1	1	100%
Category II	SCAN, Stencil, Monte Carlo, SAD, LBM, HashGPU	2-10	1-4	95%-100%
Category III	BFS	79	2	>60%

Addressing Challenge 2: Representative

- **Using architectural simulators for performing fault injections is both time-consuming and inaccurate**
  - Cannot execute applications to completion
  - Cannot model detection mechanisms accurately



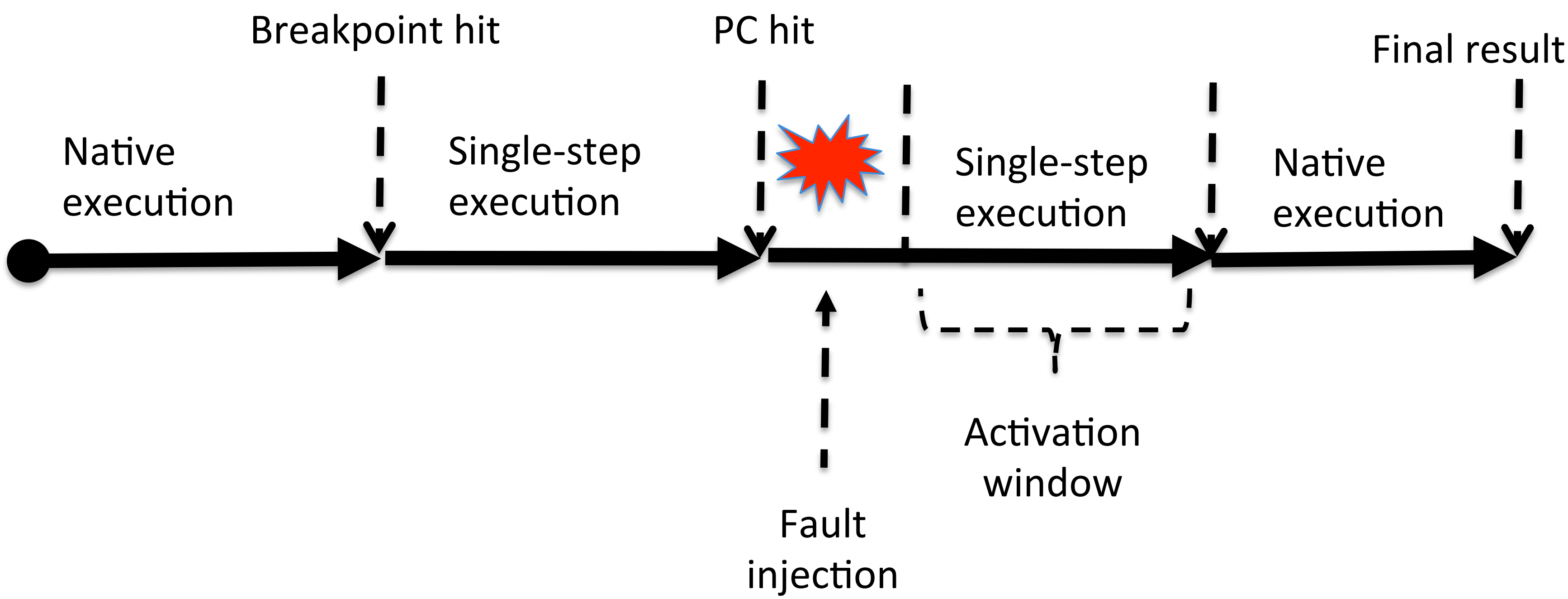
Addressing Challenge 2: Representative

- **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

- **Uniformly choose a instruction to inject fault into from all the dynamically executed instructions in the program**

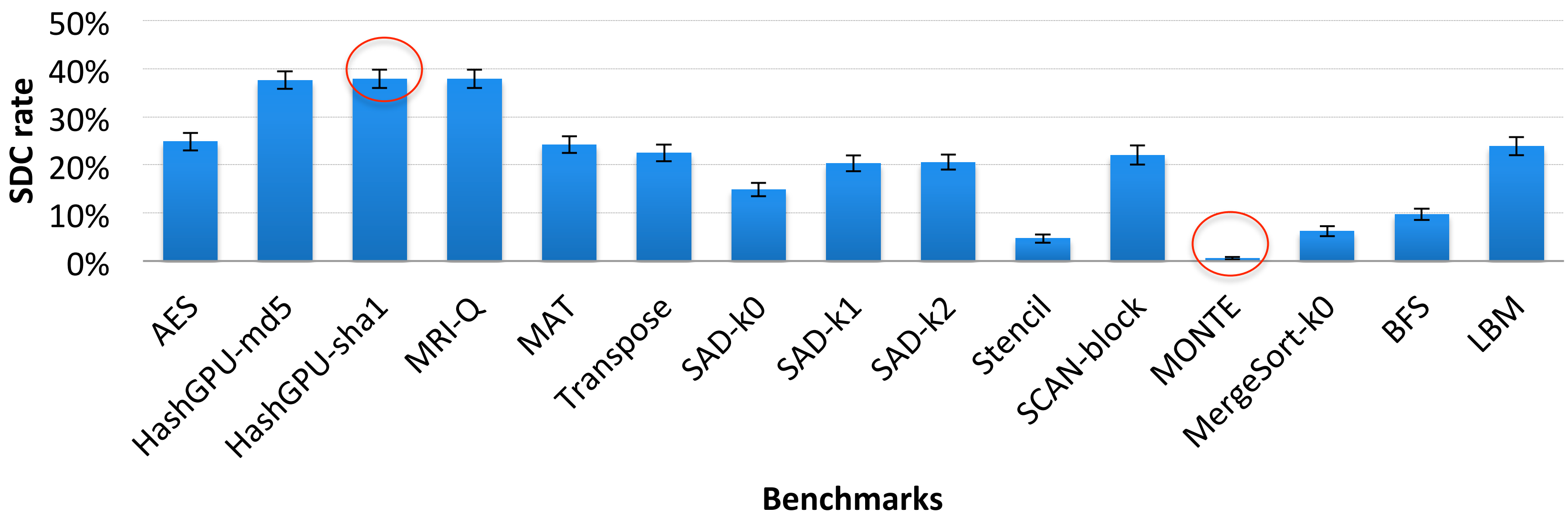


- **Only consider activated faults i.e., faults read by the system**

Experimental Setup

- **NVIDIA® Tesla C 2070/2075**
- **12 CUDA benchmarks comprising 15 kernels**
  - Rodinia, Parboil and Cuda-SDK benchmark suites
- **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 - 1

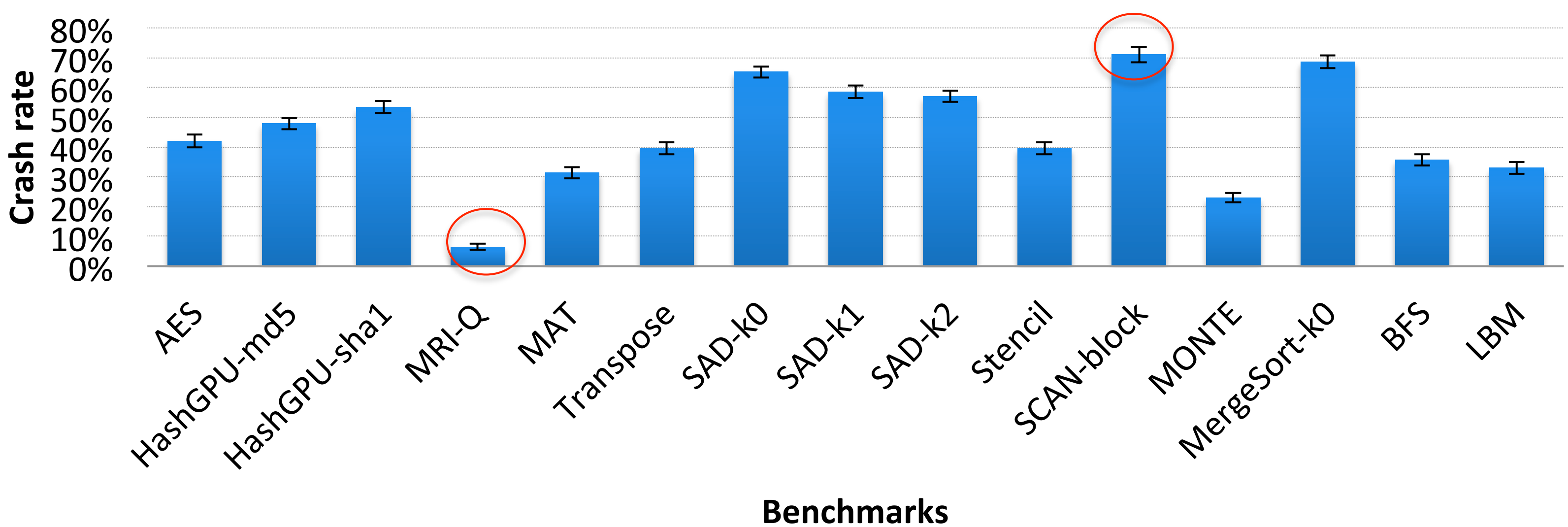


SDC Rates vary significantly across benchmarks (from 2 to 40%), which is much higher than in CPU applications (typically between 5 and 10%)

Overall Characterization Results - 2

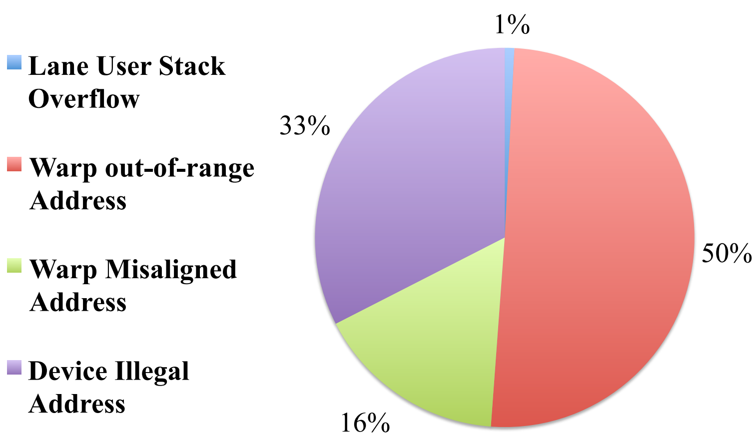
Crashes vary from 5% to 70%, across applications, which is also a much larger variation compared to CPU applications (which vary from 30 to 40%).

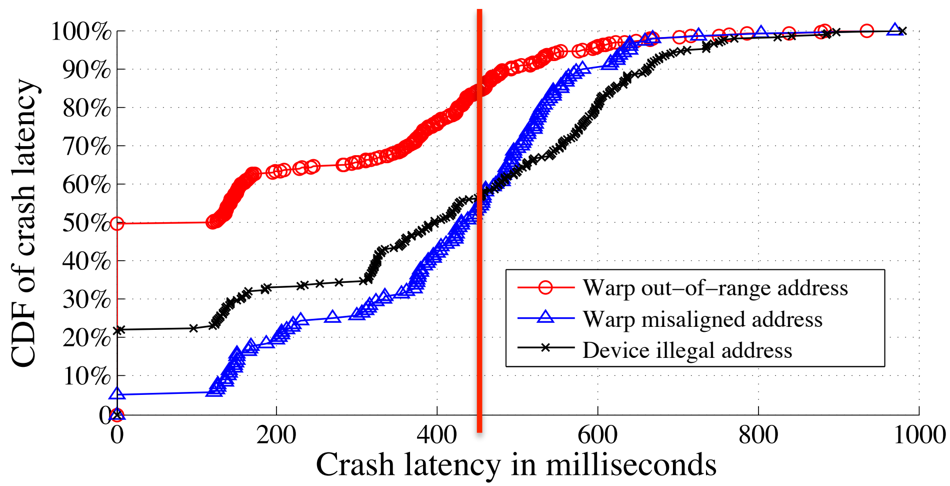
Benchmarks



Results: Crash Causes and Latency

- **Most crashes are caused by memory addresses going out of bounds and being detected by the hardware**
- **Crash latencies vary depending on type of exception, but are on the order of hundreds of milliseconds**





Hypothesis: Algorithmic Categories

- **Resilience correlated with algorithmic properties**
  - **Mapping to dwarves of parallelism [Berkeley'07]**

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
- **Crash latencies on GPUs can be much higher than CPUs**
  - Need for faster error detection in hardware
- **Correlation between algorithm and error resilience**
  - Can be used to obtain quick estimates without FI
  - Can be used to customize level of protection provided

Conclusion and Future Work

- **GPU-Qin:** Fault Injection method to systematically study GPGPU applications' error resilience
  – Understand correlations between application properties and application's error resilience

- **Future Work**
  – Understand GPU hardware detection mechanisms
  – Extend to OpenCL programs, other GPUs
  – Software mechanisms to protect the application

Thank you !

More details: Read our ISPASS'14 paper

"*GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications*", **Bo Fang**, Karthik Pattabiraman, Matei Ripeanu, Sudhanva Gurumurthi, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14), Mar 23-25, 2014.

GPU-Qin is available for download (BSD style license)

https://github.com/DependableSystemsLab/GPU-Injector