Fine-Grained Characterization of Faults Causing Long Latency Crashes in Programs

Guanpeng Li, Qining Lu and Karthik Pattabiraman

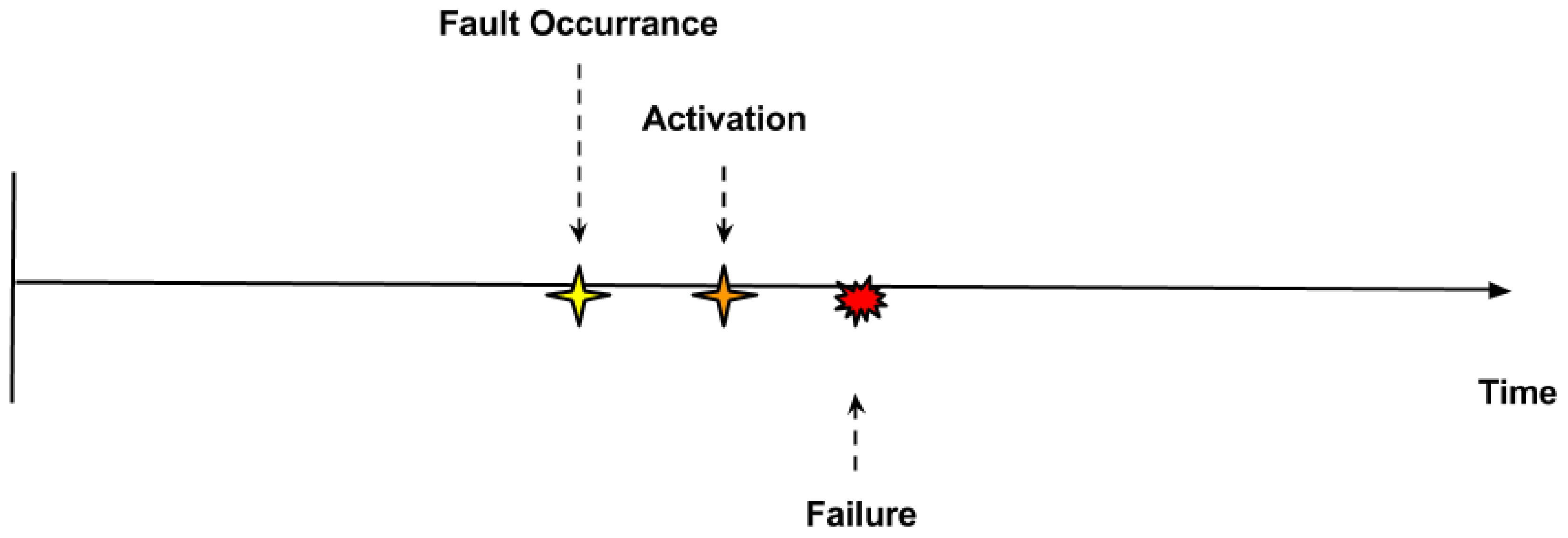
University of British Columbia, Canada

Soft Errors

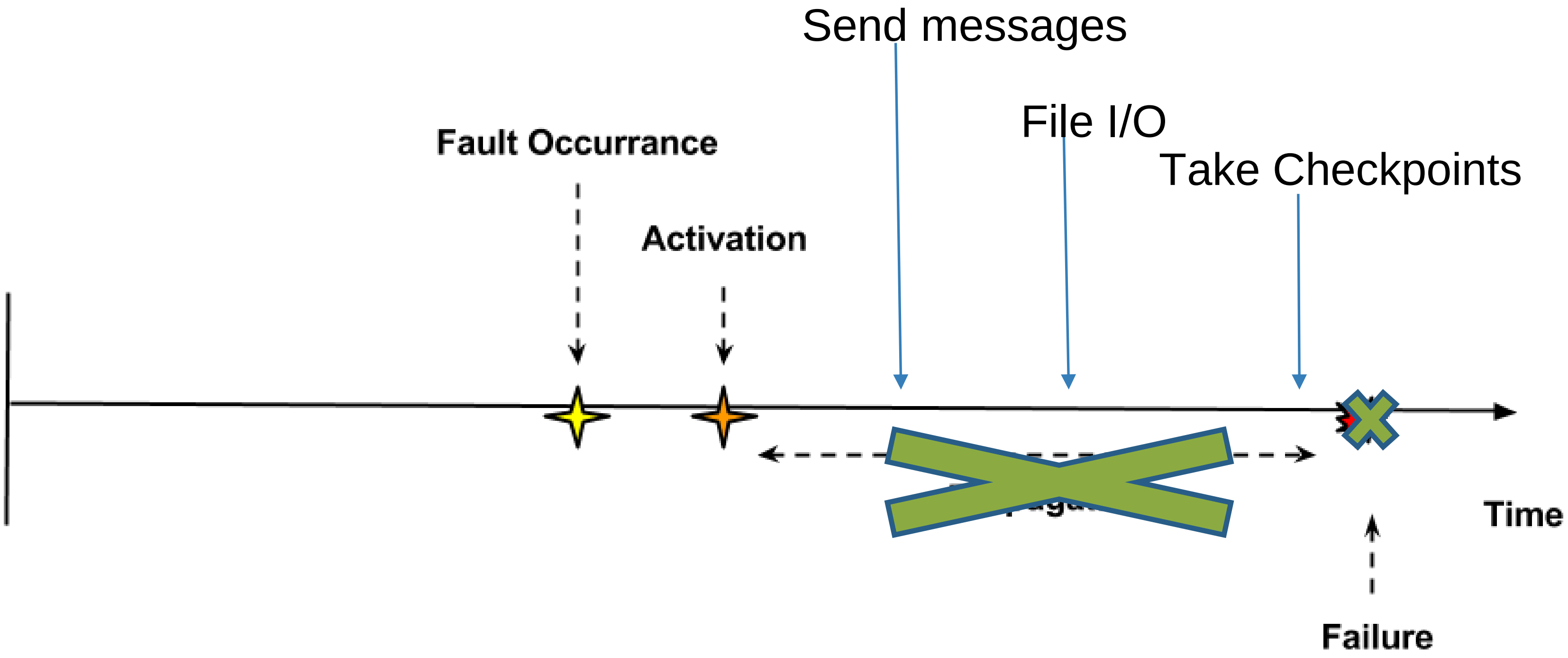
Soft error rate will increase by nearly an order of magnitude as chip feature sizes decrease from 22nm to 14nm.

[Feng et. al., ASPLOS'10]

Fail-stop Assumption



But, in reality ...



Traditional Solutions

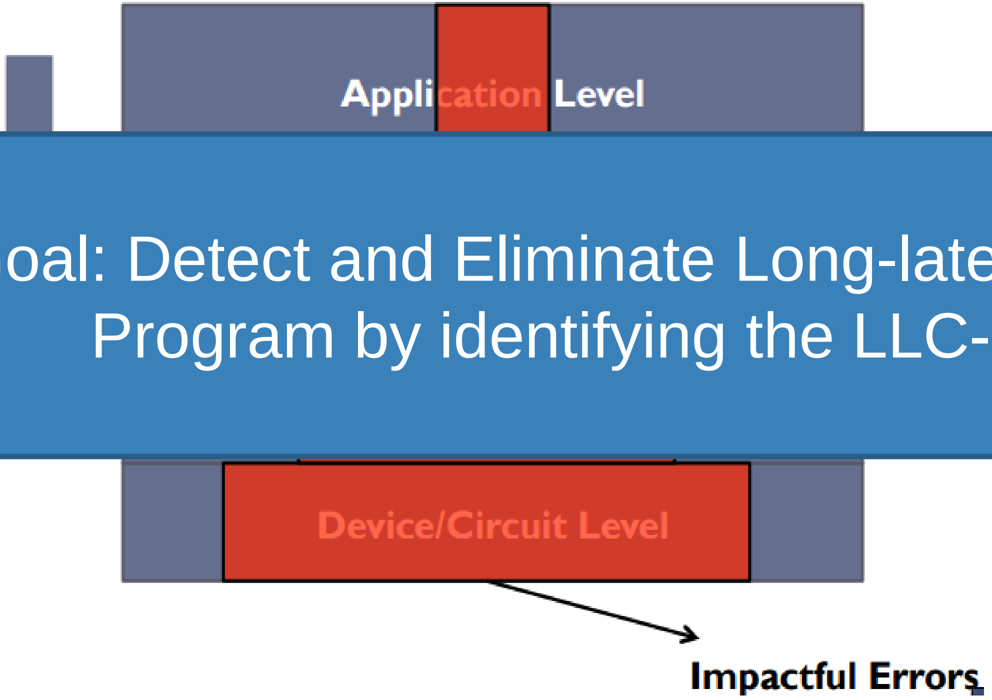
- **Duplication**

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

- **Guard-banding**

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

Why Software ?



Goal: Detect and Eliminate Long-latency Crashes(LLCs) in Program by identifying the LLC-causing locations

Challenges

Statistical Fault Injection

- Good for resiliency characterization
- Takes long time to find LLCs

What we do

Code patterns leading to LLC fall into very few dominant patterns

Static analysis to identify the patterns

Selective sampling to filter out false-positives

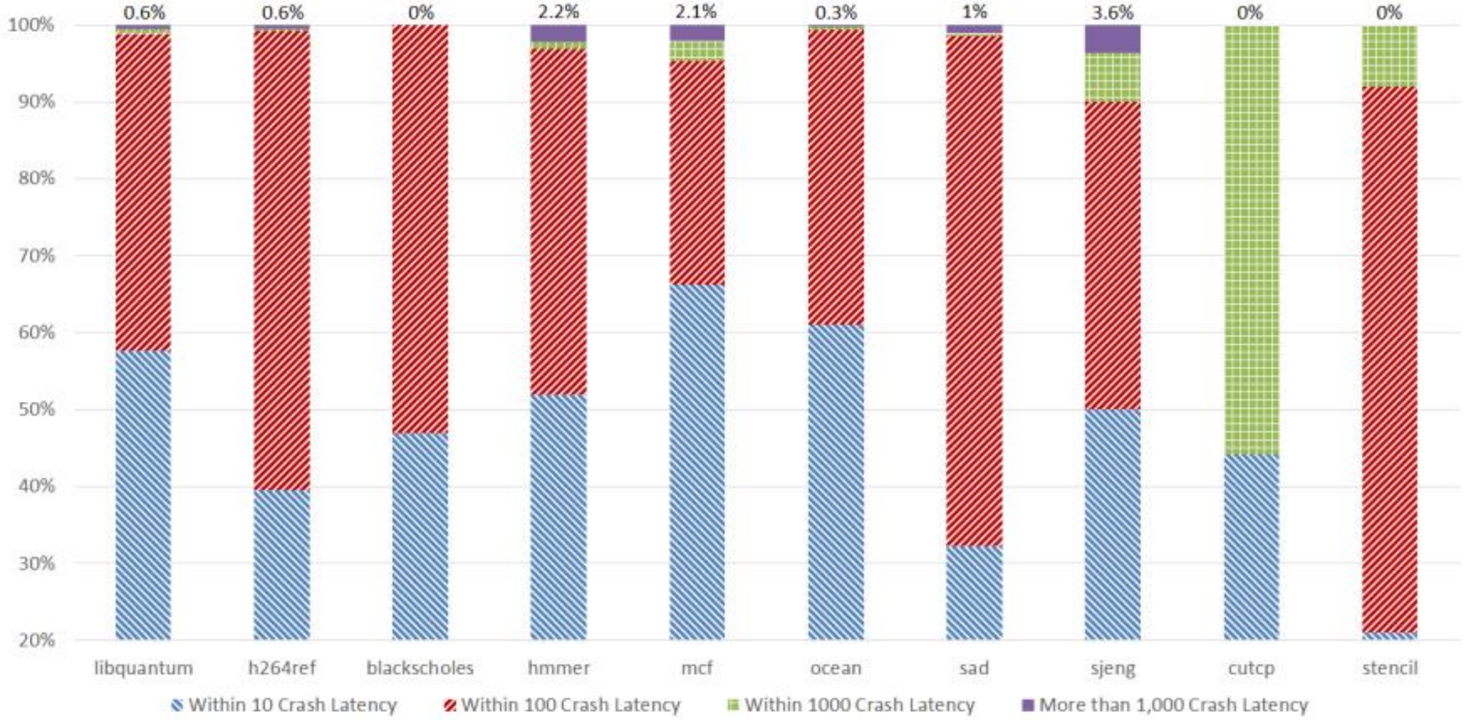
Initial Fault Injection Study

- Choose 5 from 10 benchmark applications
- 1,000 random fault injections per application
- 1 fault injection per run – single bit flip

Fault Model

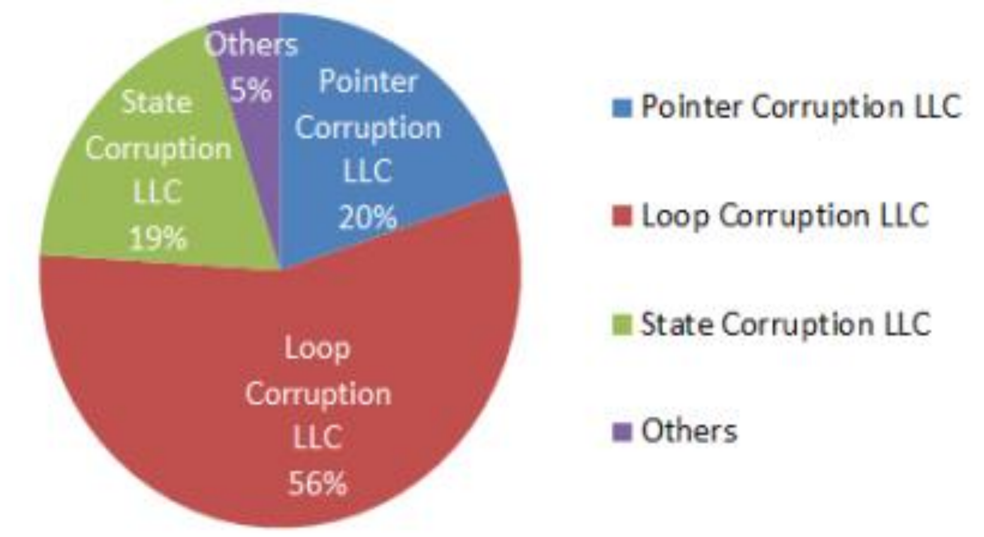
- Faults occur in computational components or load/store units in CPU
- Assume memory and cache - ECC protected
- LLVM Fault Injector (LLFI) [Wei – DSN'14]

Propagation Latency (dynamic insns)



Propagation latency is application-specific

Patterns Leading to LLCs



What we do

Code patterns leading to LLC fall into very few dominant patterns

CrashFinder Static

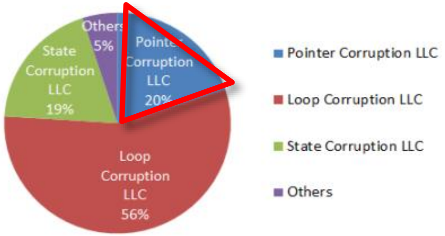
Static analysis to identify the patterns

Selective sampling to filter out false-positives

Pointer Corruption LLC: Example

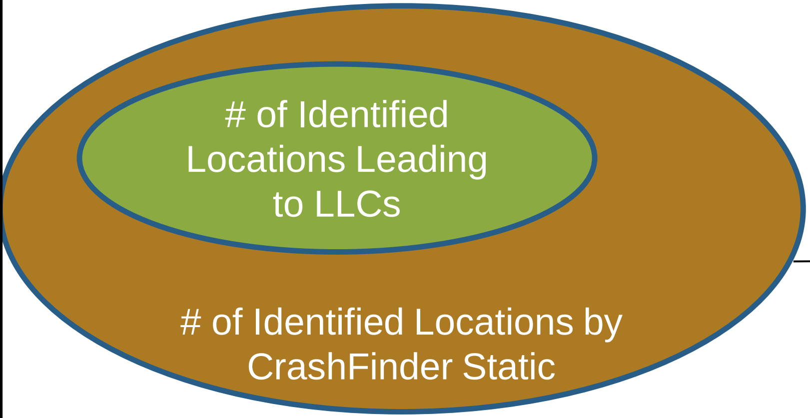
```
static unsigned int state[N+1];
static unsigned int *next;
...
unsigned int reloadMT(void)
{
  ...
  register unsigned int *p0 = state;
  next = state+1;
  ...
  *p0++ = *pM++ ^ ... ;
  ...
}
...
unsigned int randomMT(void)
{
  unsigned int y;
  ...
  y  = *next++;
  ...
}
...
```

[From sjeng program]



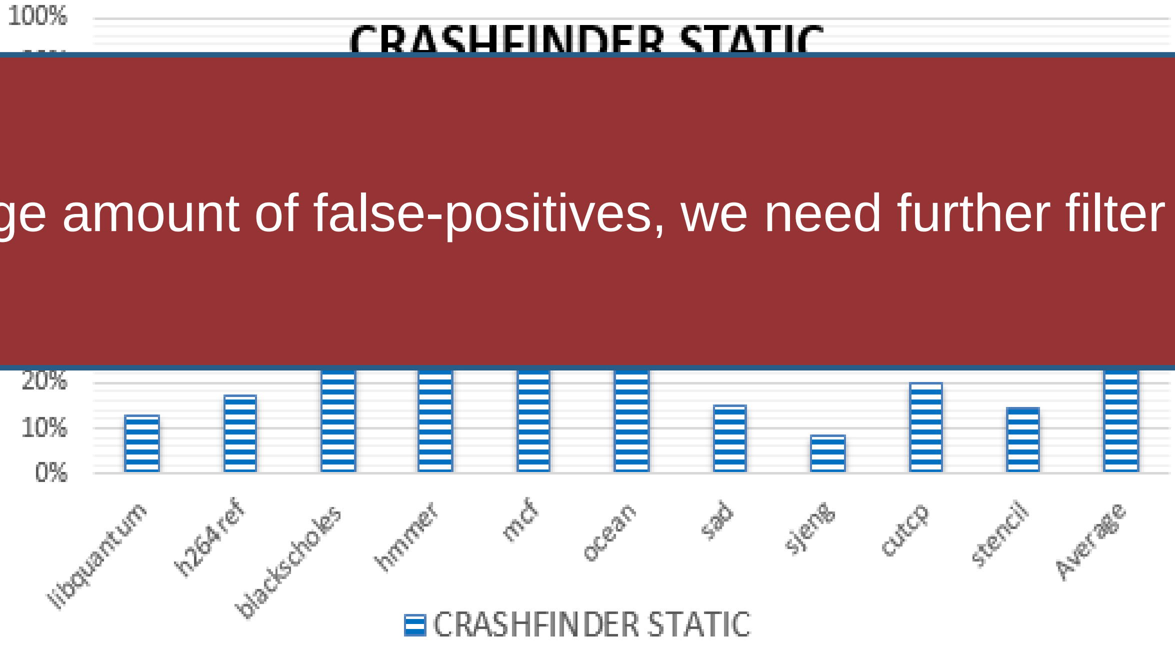
Precision

- 200 random fault injections on each static location identified by the technique
- 10 applications from 4 benchmark suites



$$\frac{\textbf{True Positives}}{\textbf{True Positives + False Positives}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$



Large amount of false-positives, we need further filter them out!

What we do

CrashFinder Static

Code patterns leading to LLC fall into very few dominant patterns → Static analysis to identify the pattterns → Selective sampling to filter out false-positives

2 Heuristics:
H1 & H2

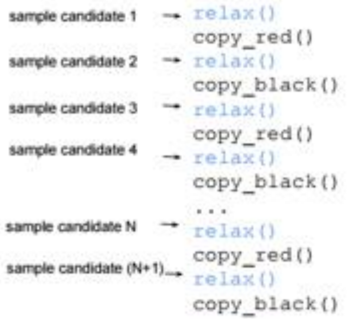
H1: Instruction Sampling

Similar behavior in similar control flow

More efficient to sample by unique function call sequence

```
void multig(long my_id){
  ...
  while ((!flag1) && (!flag2)) {
    ...
    relax();
    copy_red();
    relax();
    copy_black();
    ...
    }
}

void relax(){
  ...
  for (...) {
    ...
    t1a = (double *) t2a[i];
    ...
  }
}
```



[From ocean program]

H2: Bit Sampling

What we do

CrashFinder Static

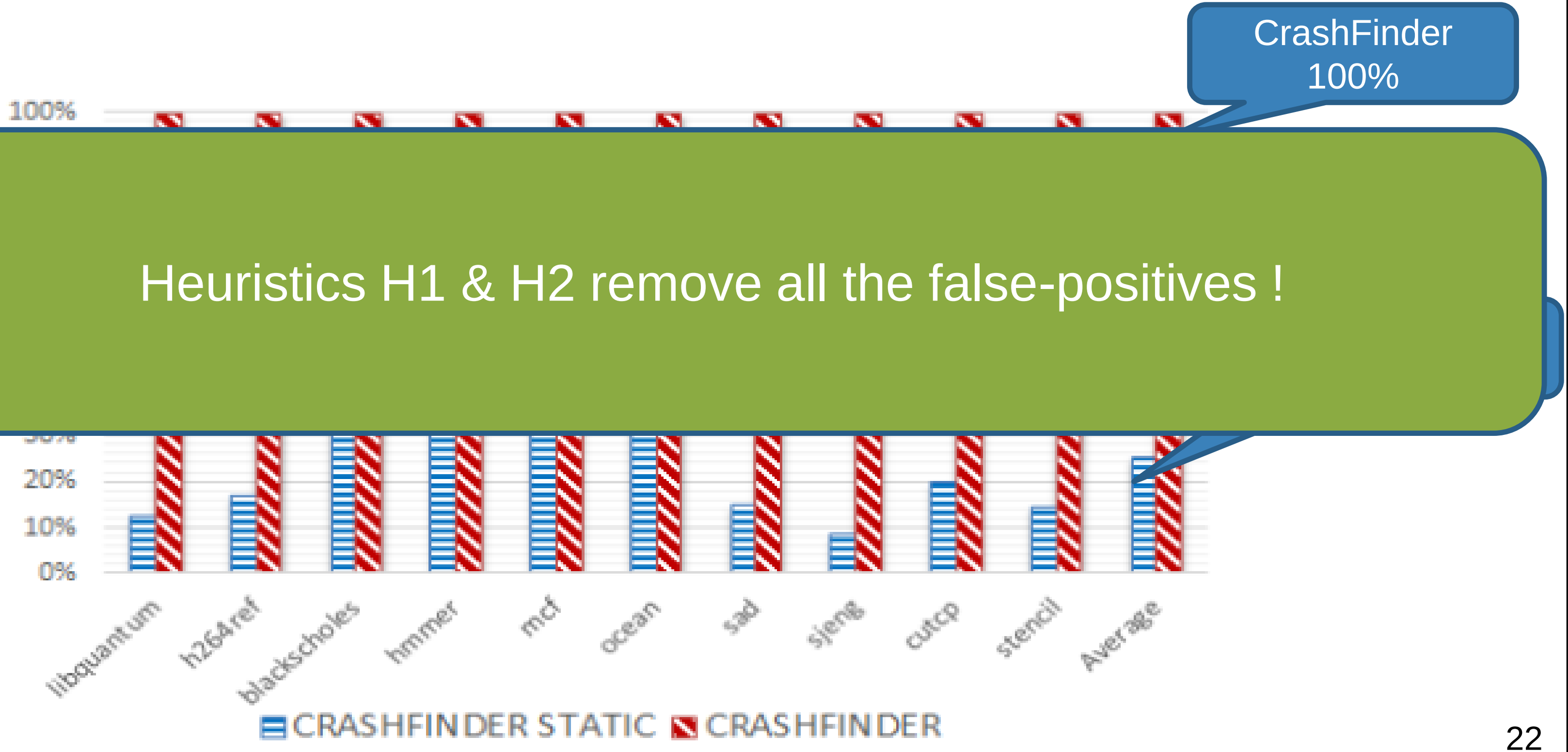
Code patterns leading to LLC fall into very few dominant patterns → Static analysis to identify the patterns → Selective sampling to filter out false-positives

2 Heuristics: H1 & H2

CrashFinder

- **LLVM compiler pass**
- **No annotation required - Fully automated**
- **Supports C and C++ programs**

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives + False Positives}}$$

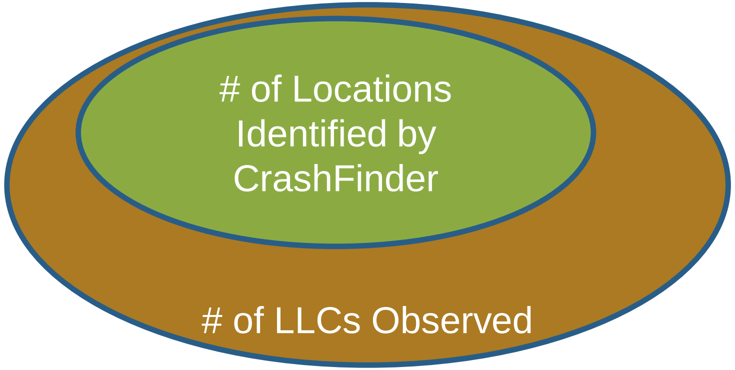


Heuristics H1 & H2 remove all the false-positives !

Recall

Experiment

- 3,000 random fault injections on each application
- Total of 10 benchmarks

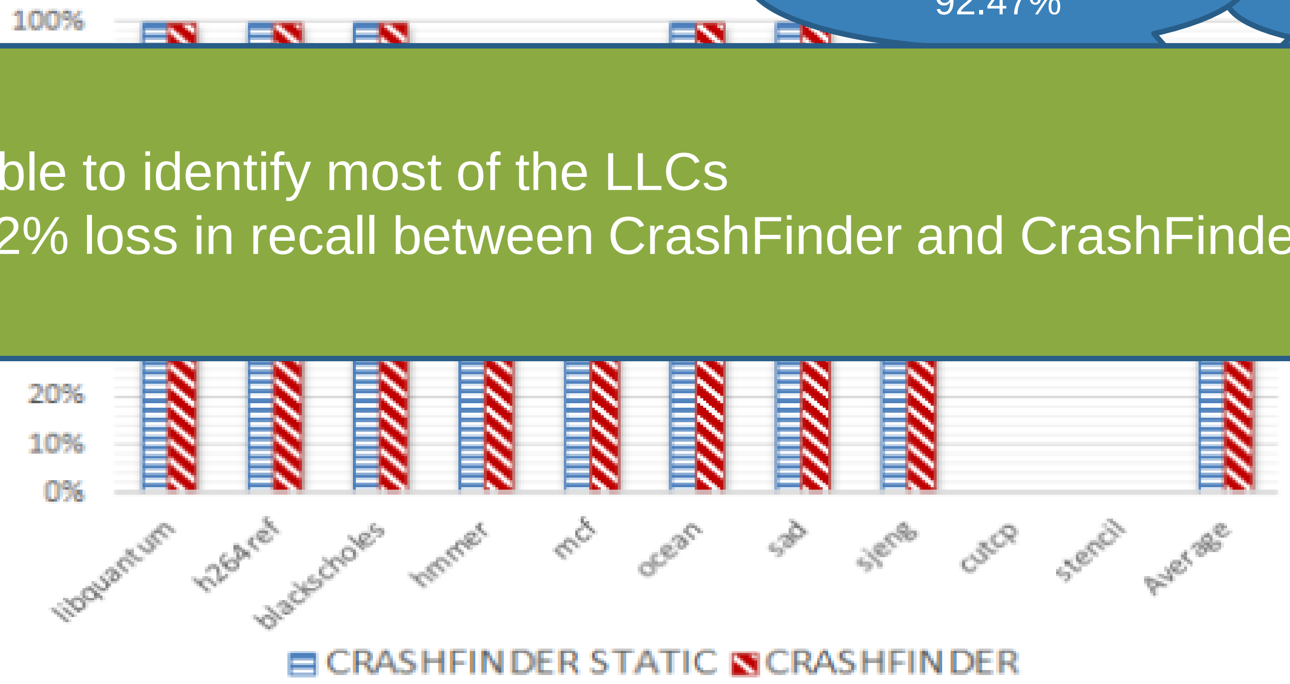


$$\frac{\textbf{True Positives}}{\textbf{True Positives + False Negatives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}$$

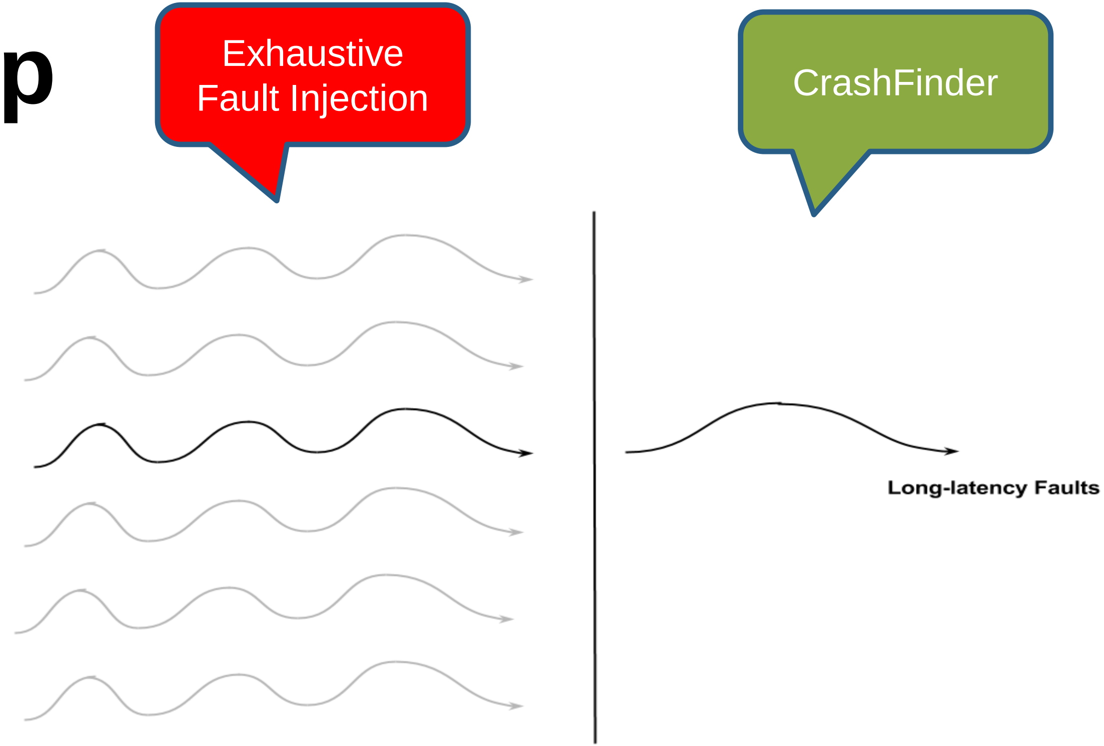
CrashFinder Static
92.47%

CrashFinder
90.14%



- Able to identify most of the LLCs
- ~2% loss in recall between CrashFinder and CrashFinder Static

Speed Up

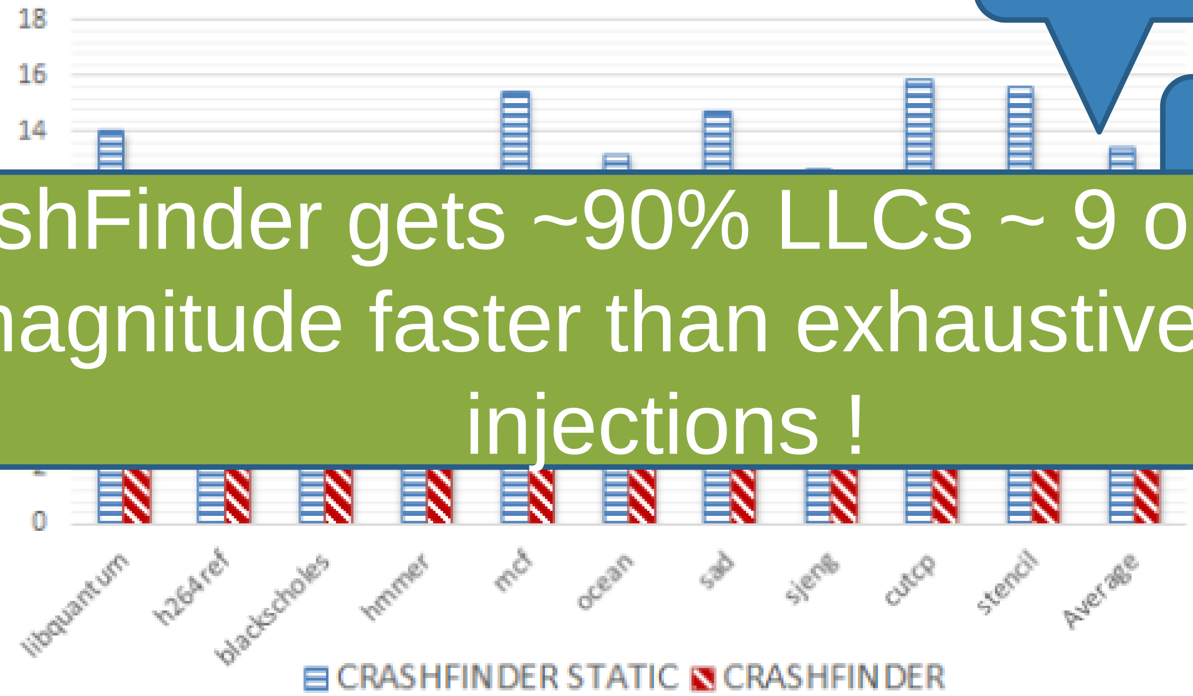


$$\text{Speed Up} = \frac{\text{Time taken by CrashFinder}}{\text{Time taken by exhaustive fault injection}}$$

Speed Up: Orders of Magnitude

CrashFinder Static
Time: ~0.5 min
Speedup : ~13 OoM

CrashFinder



CrashFinder gets ~90% LLCs ~ 9 orders of magnitude faster than exhaustive fault injections !

Implications: Costs and Benefits

Performance Overhead [under submission]

- ~5% by selective duplication of LLC causing locations' backward slices

Availability [under submission]

- Avoids ~96% of checkpoint corruptions
- About 8 times reduction in unavailability
  (unavailability = 1 - availability)

Related Work

- Long-latency faults have been observed, but noone has identified patterns leading to them [Chandra 2002] [Gu 2003] [Yim 2009]

- Relyzer [Hari 2011], SDCTune [Lu 2014] reduces fault injection space for SDCs
- Non-trivial to extend for LLCs which are much rarer

Summary

- Long-latency crashes (LLCs) fall into 3 dominant code patterns, which can be identified thro’ static analysis
- Heuristics used in CrashFinder works well with ~90% recall and 100% precision (i.e., no false positives)
- Speedup of more than 9 orders of magnitude compared to exhaustive fault injection (current state of the art)

gpli@ece.ubc.ca

https://github.com/DependableSystemsLab/Crashfinder