Modeling Hardware Error Propagation in Programs for Low-Cost Dependability

Guanpeng(Justin) Li

Qining Lu, Bo Fang,

Karthik Pattabiraman

Siva Hari, Michael Sullivan, Timothy Tsai

Chen-Yong Cher, Meeta Gupta, Jude Rivers, Pradip Bose

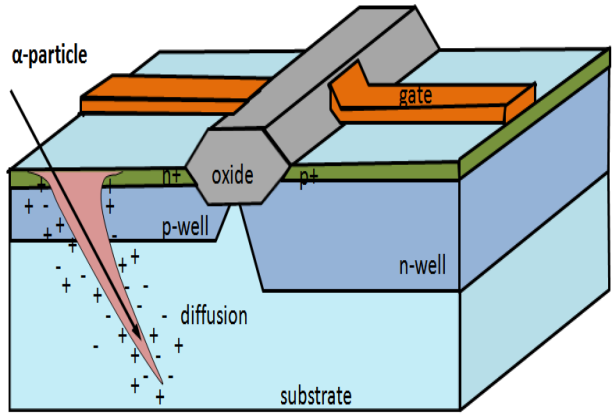
My Research

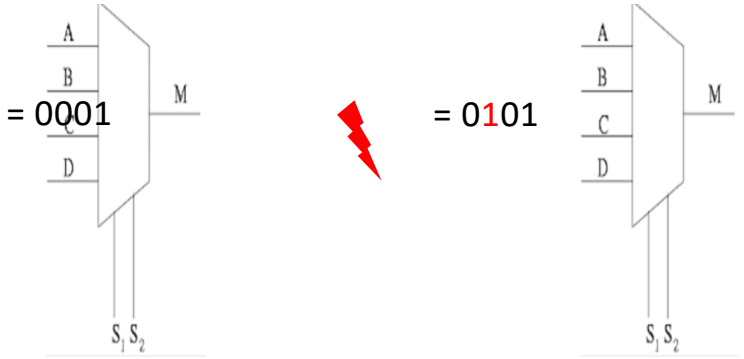
- **Building fault-tolerant and secure software systems**

- **Three areas**
  - Software error resilience
    [DSN'12][DSN'13][DSN'14][CASES'14][DSN'15][SC'16][DSN'16][SC'17][DSN'17][DSN'18A][DSN'18B]
  - Web applications' reliability
    [ICSE'14A][ICSE'14B][ICSE'15][ICSE'16][ASE'14][ASE'15][ASE'17][ICSE'18]
  - IoT devices security [FSE'17][ACSAC'16][EDCC'15]

- **This talk: Software error resilience techniques**

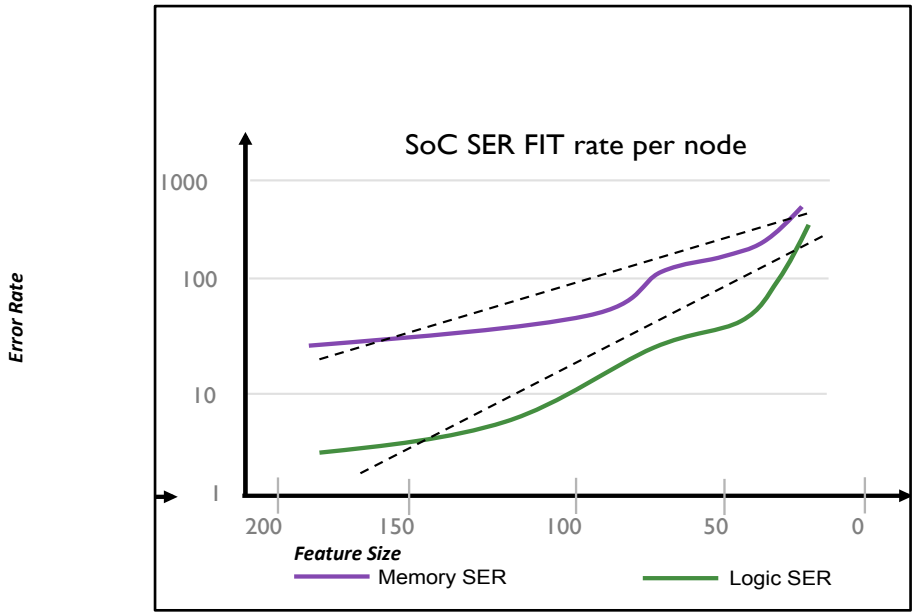
Motivation: Soft Errors





Motivation: Soft Errors

- **Soft errors becoming more common in processors**

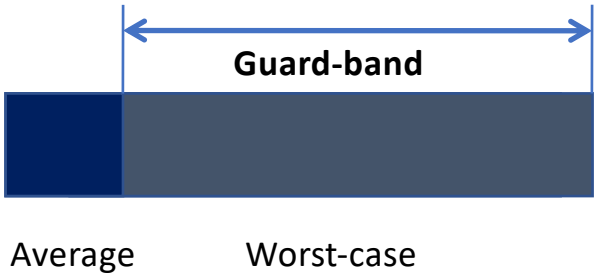


[Chandra et. al., DATE'14]

Soft Errors: Traditional Solutions

- **Guard-banding**

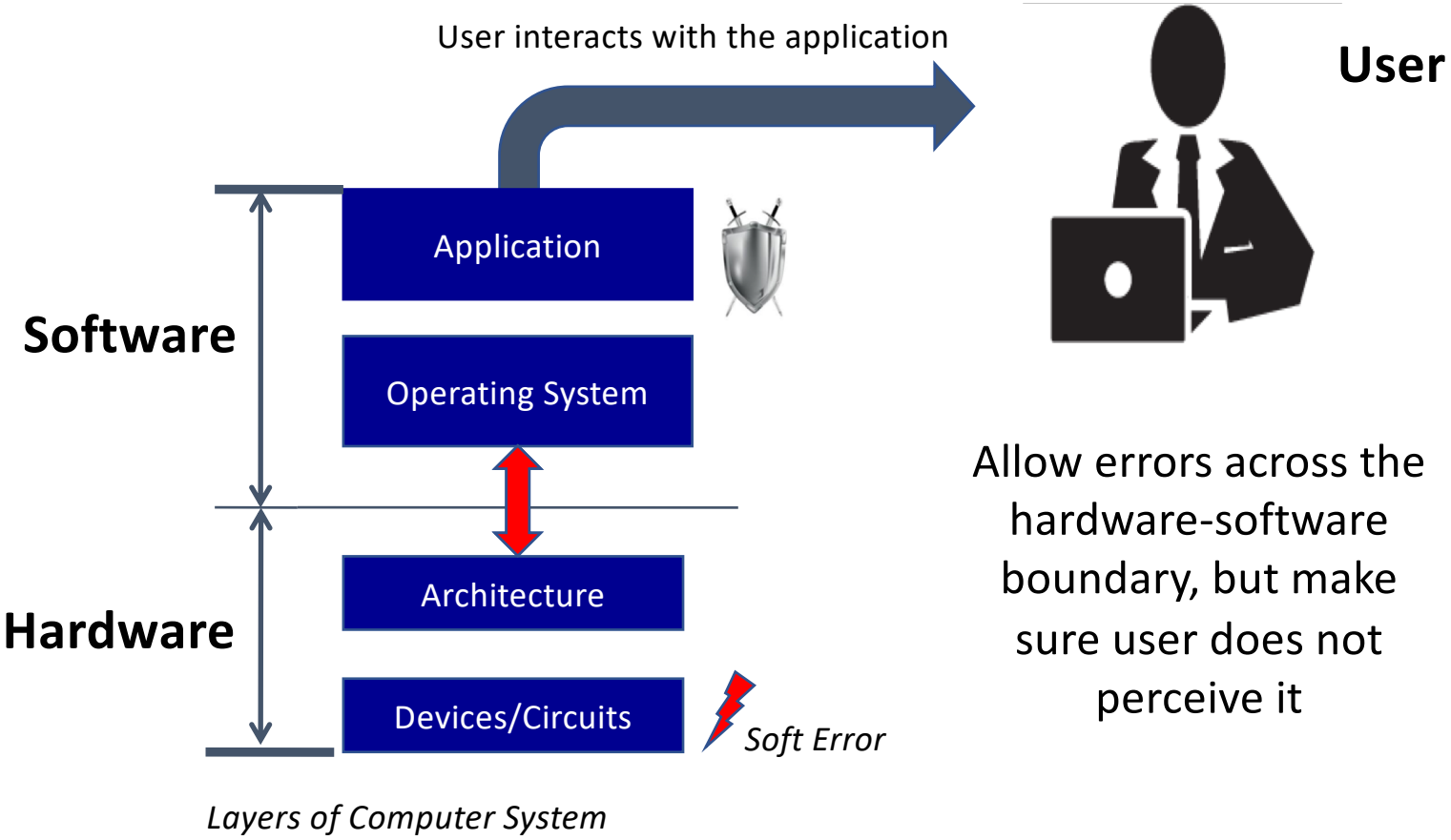
Guard-banding wastes power as gap between average and worst-case widens due to variations



- **Duplication**

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

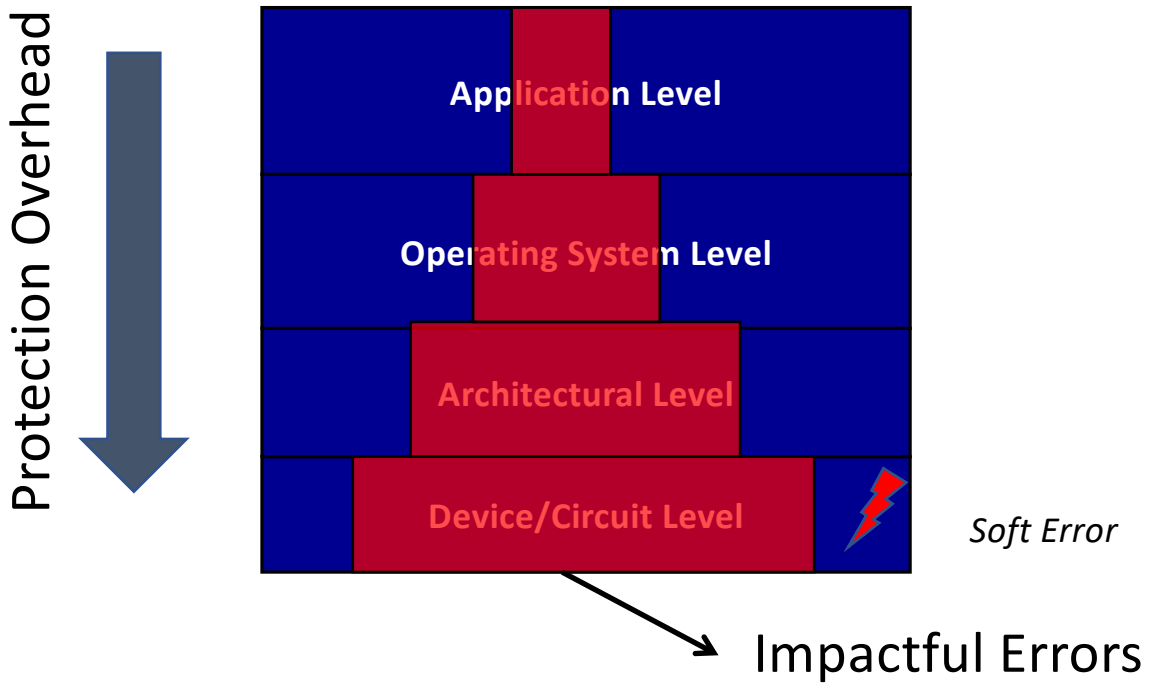
An alternative approach



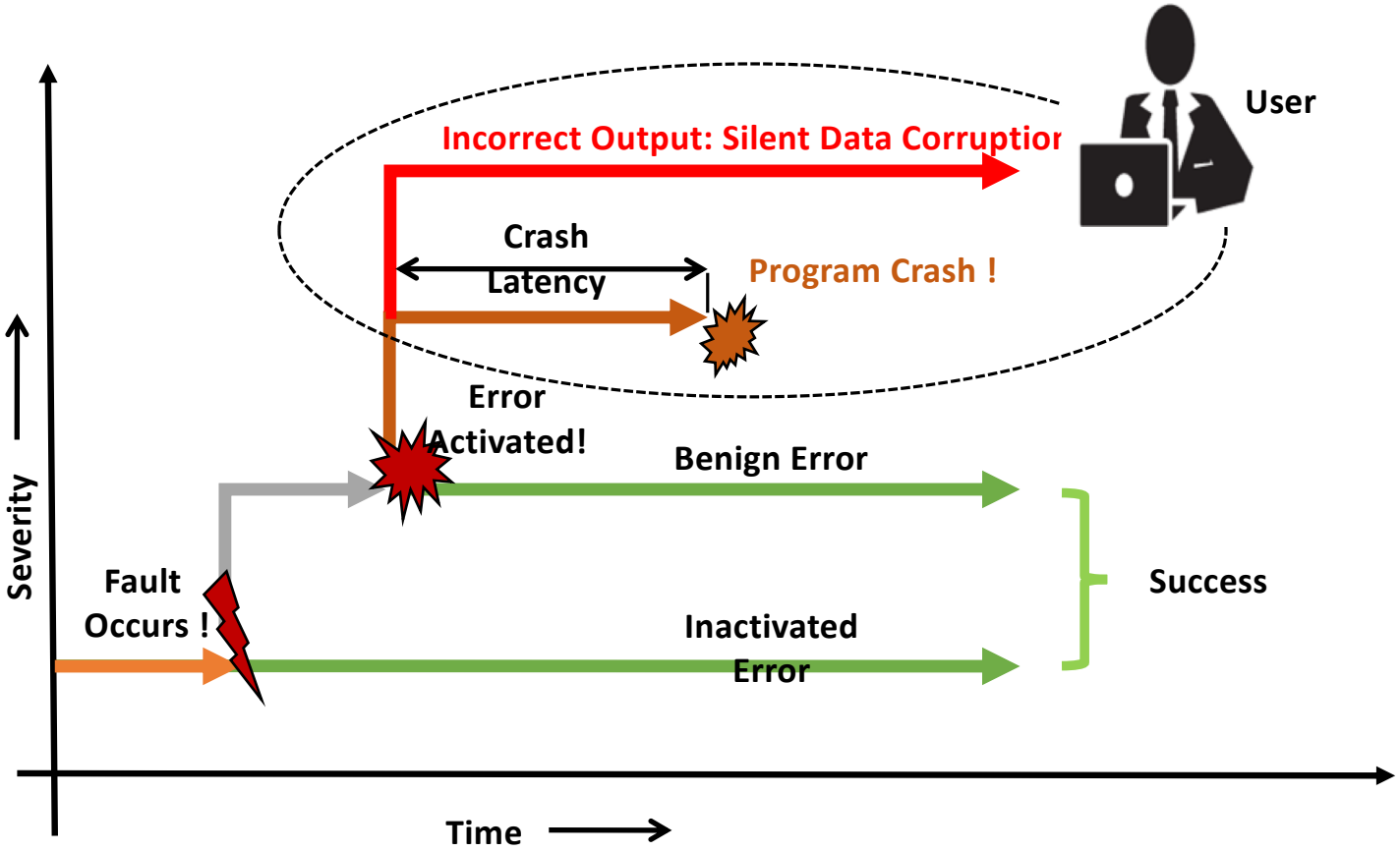
Layers of Computer System

Allow errors across the hardware-software boundary, but make sure user does not perceive it

Why does this approach work ?

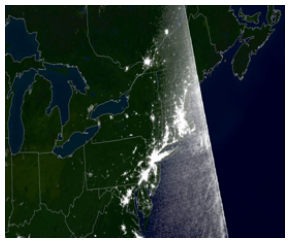


Error Propagation in Programs



Consequences of Error Propagation

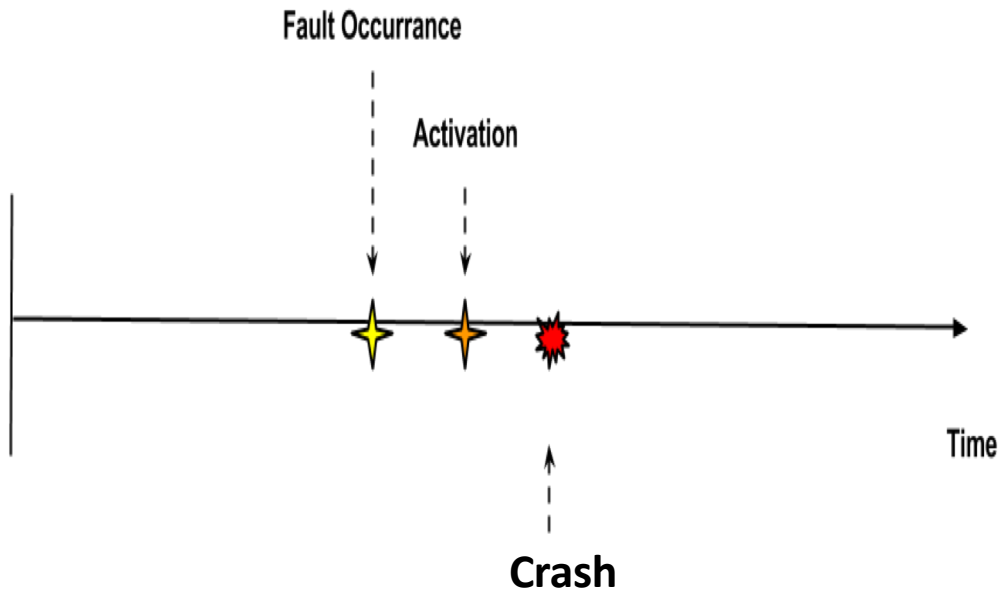
- **Well-known incidents**
  - Northeastern power outage of 2003
  - AWS outage of 2008 - single bit flip propagation
- **Sudden acceleration in Toyota cars causes fatal accidents**
  - Paid $1.2B for avoiding prosecution [2013]



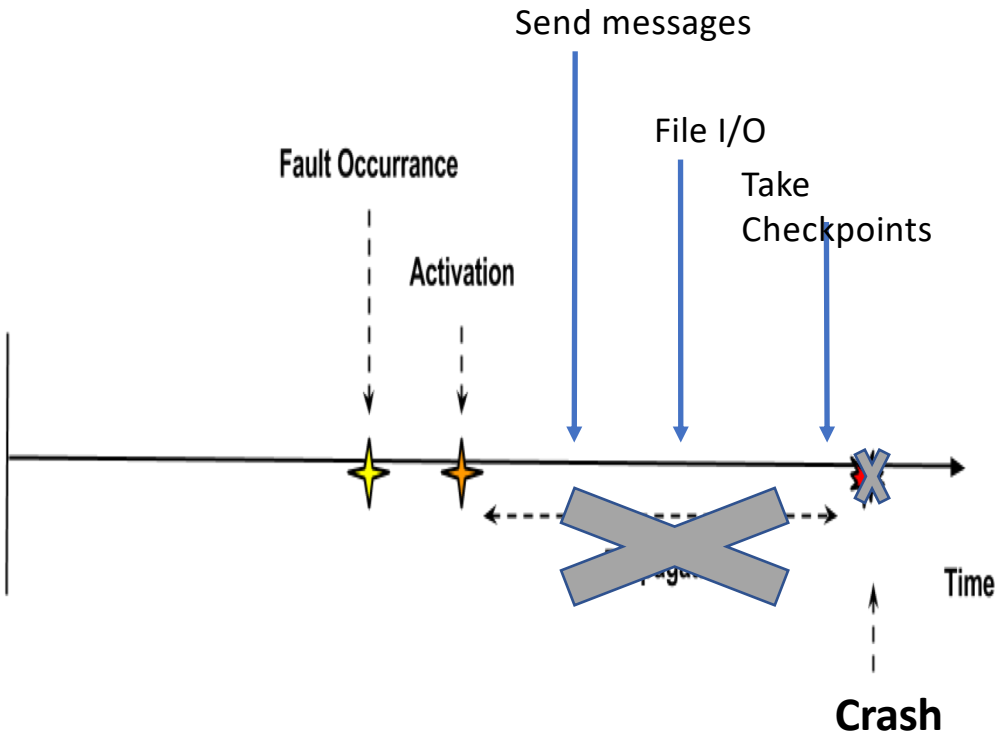

Outline

- Motivation and Goals
- Empirical Studies of Error Propagation [DSN'15] [ISSRE'15][SC'16]
- Models of Error Propagation [CASES'14] [TECS] [DSN'16] [DSN'18]
- Future Work and Conclusions

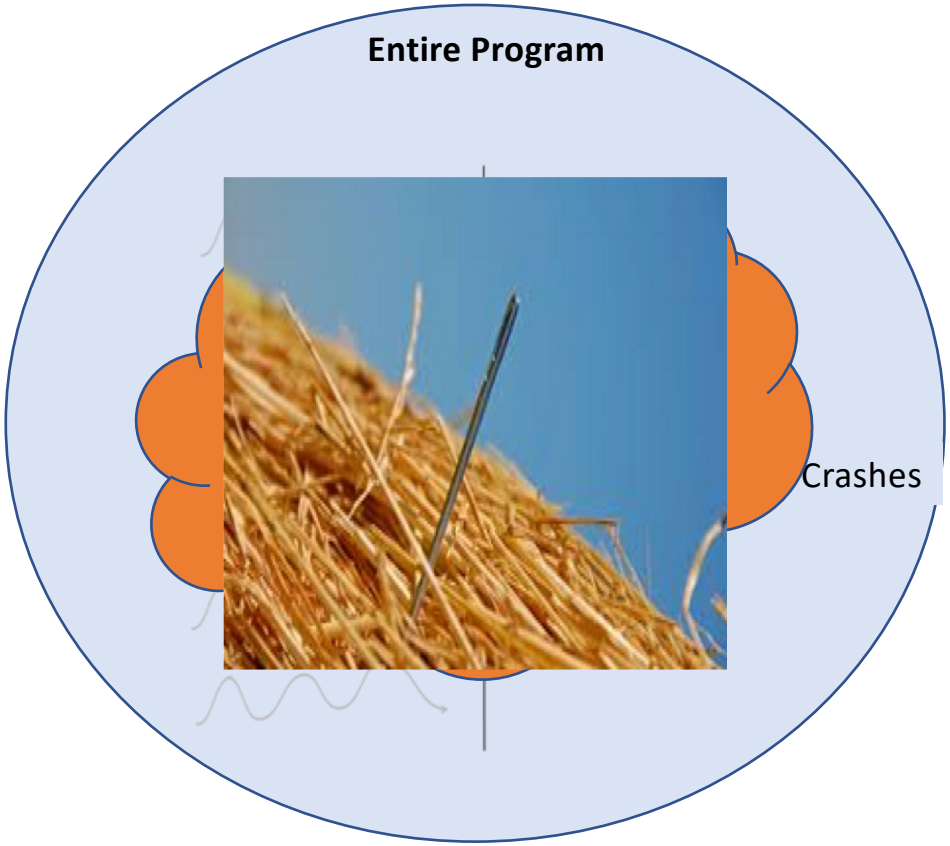
CrashFinder: Fail-Stop Assumption



CrashFinder: But, in reality ...



CrashFinder: Challenges



CrashFinder: Our Approach

Long-latency crashes fall into a few dominant patterns

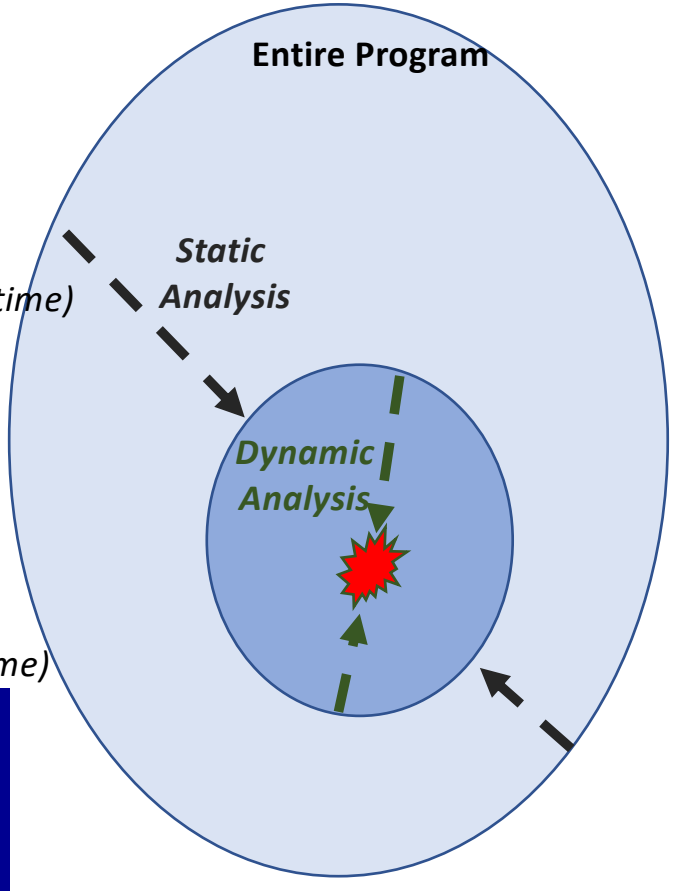
Static Analysis (Compile-time)

Code patterns leading to long-latency crashes

Static analysis to identify the patterns

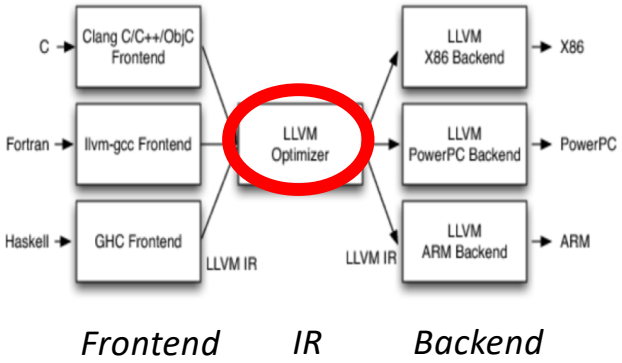
Dynamic Analysis (Runtime)

Selective sampling to filter out false-positives



LLVM Fault Injector (LLFI)

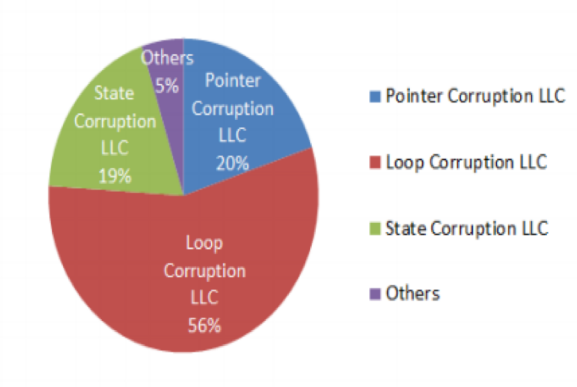
- Benefits at LLVM IR [DSN'14]
  - Inject into specific instructions, variables and code constructs
  - Easy to study propagation of errors and map result back to source code
- Also extended to GPUs [SC'16]



CrashFinder: Initial Fault Injection Study

- **Crash latency**
  - Majority < 100 instructions
- **Long latency crashes (LLCs)**
  - Up to 3.6% among total crashes
  - Latencies range from a few thousand to million instructions

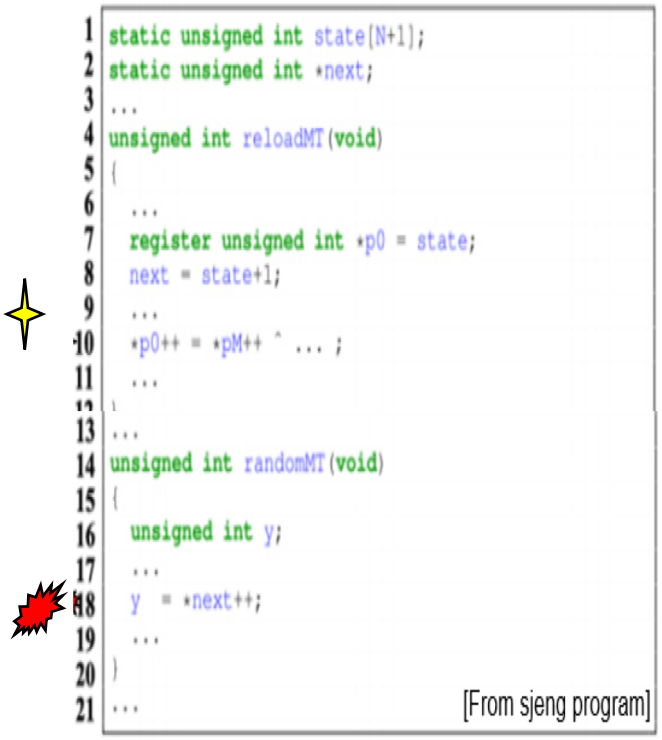
Most long-latency crashes are caused due to a few dominant code patterns



Code Patterns

CrashFinder: Static and Dynamic Analysis

Static Analysis:



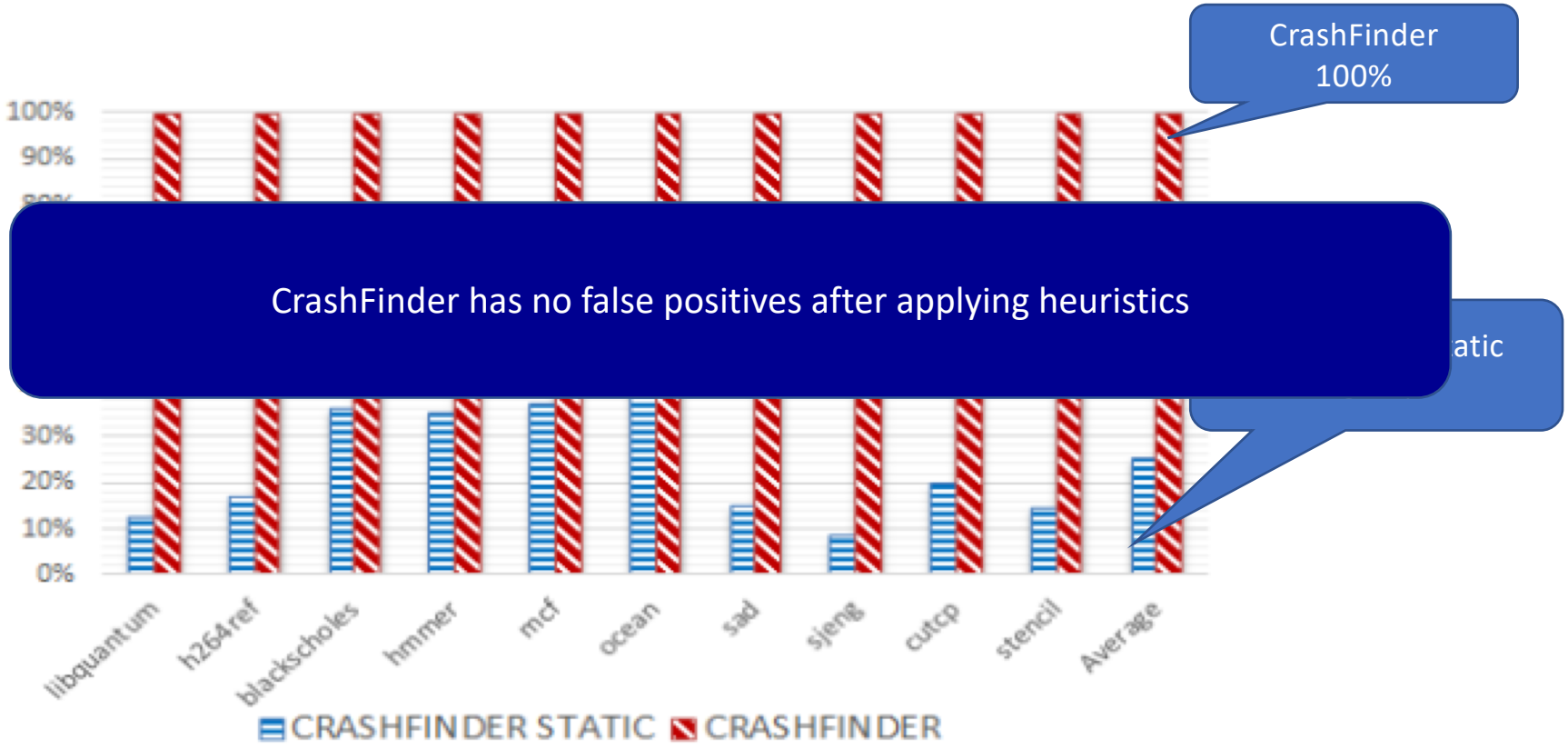
```
1  static unsigned int state[N+1];
2  static unsigned int *next;
3  ...
4  unsigned int reloadMT(void)
5  {
6    ...
7    register unsigned int *p0 = state;
8    next = state+1;
9    ...
10   *p0++ = *pM++ ^ ... ;
11   ...
12 }
13 ...
14 unsigned int randomMT(void)
15 {
16   unsigned int y;
17   ...
18   y  = *next++;
19   ...
20 }
21 ...
```

[From sjeng program]

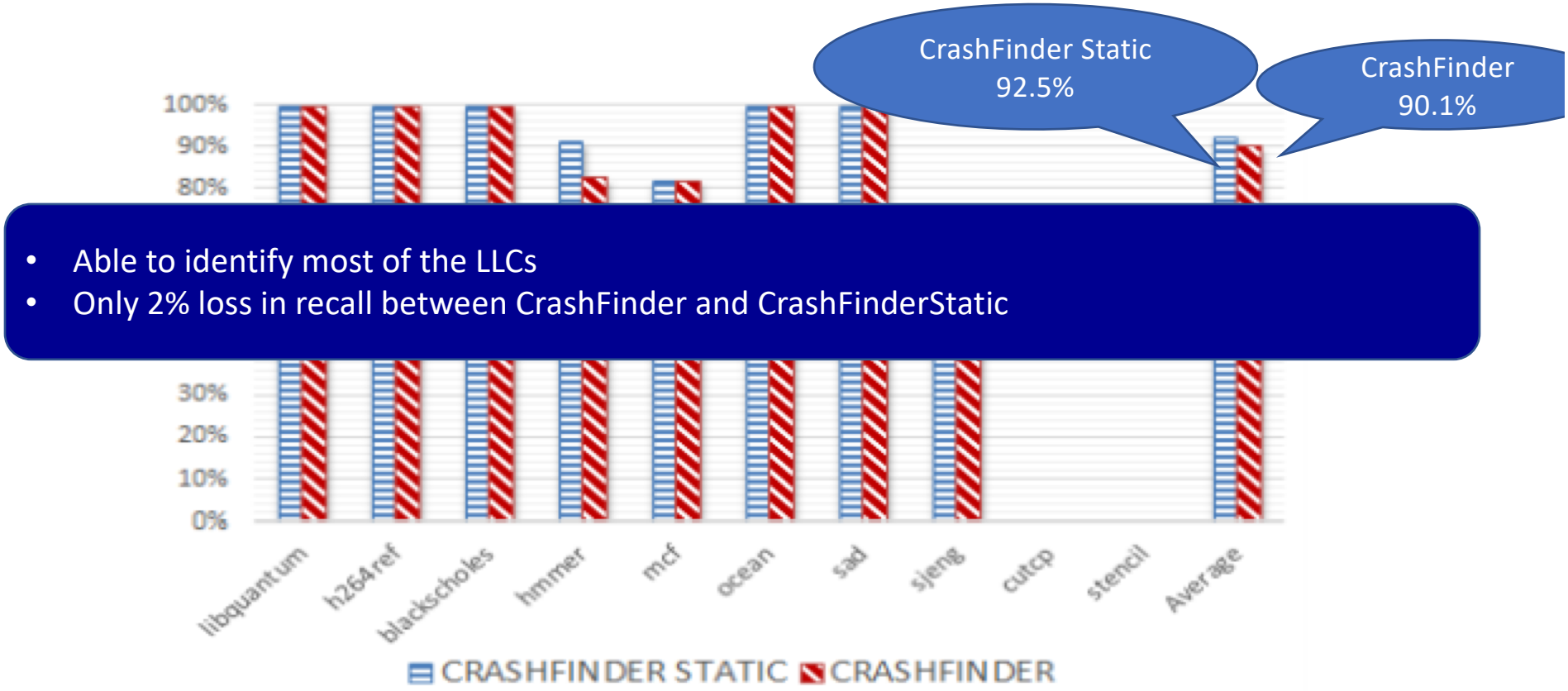
Dynamic Analysis:

- Heuristic
  - Similar control-flow path leads to similar error propagation
- Selective sampling for fault injections
  - Filter out false-positives

CrashFinder: Results (Precision)



CrashFinder: Results (Recall)



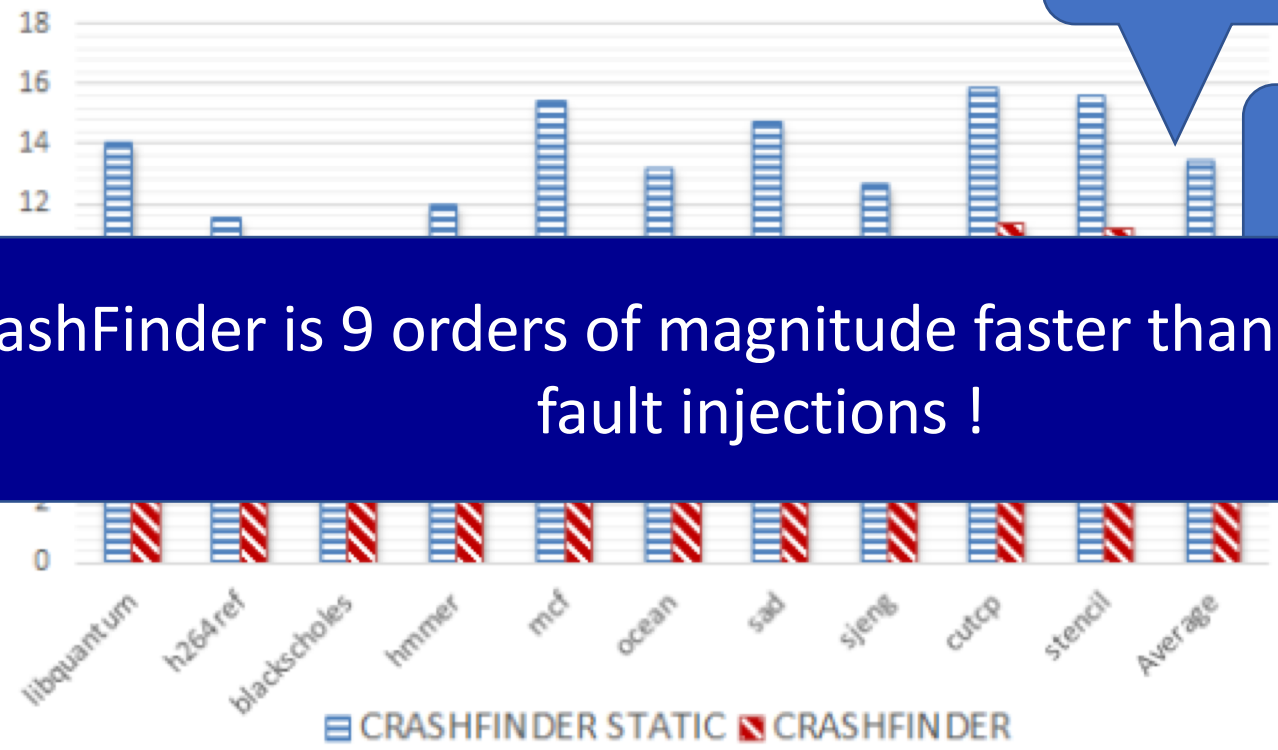
- Able to identify most of the LLCs
- Only 2% loss in recall between CrashFinder and CrashFinderStatic

CrashFinder: Fault Injection Spa

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

CrashFinder
Time: ~4 days
Space : ~9 OoM

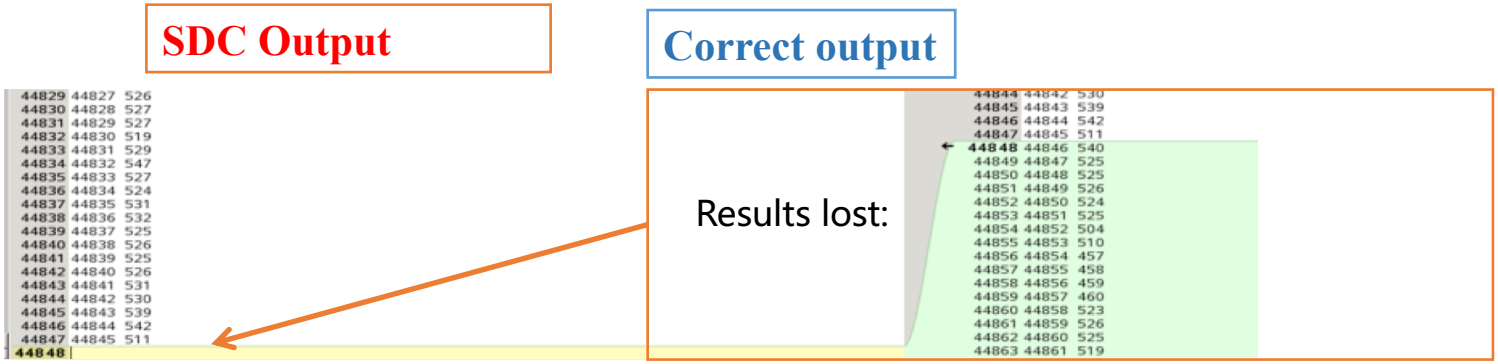
CrashFinder is 9 orders of magnitude faster than exhaustive fault injections !



Outline

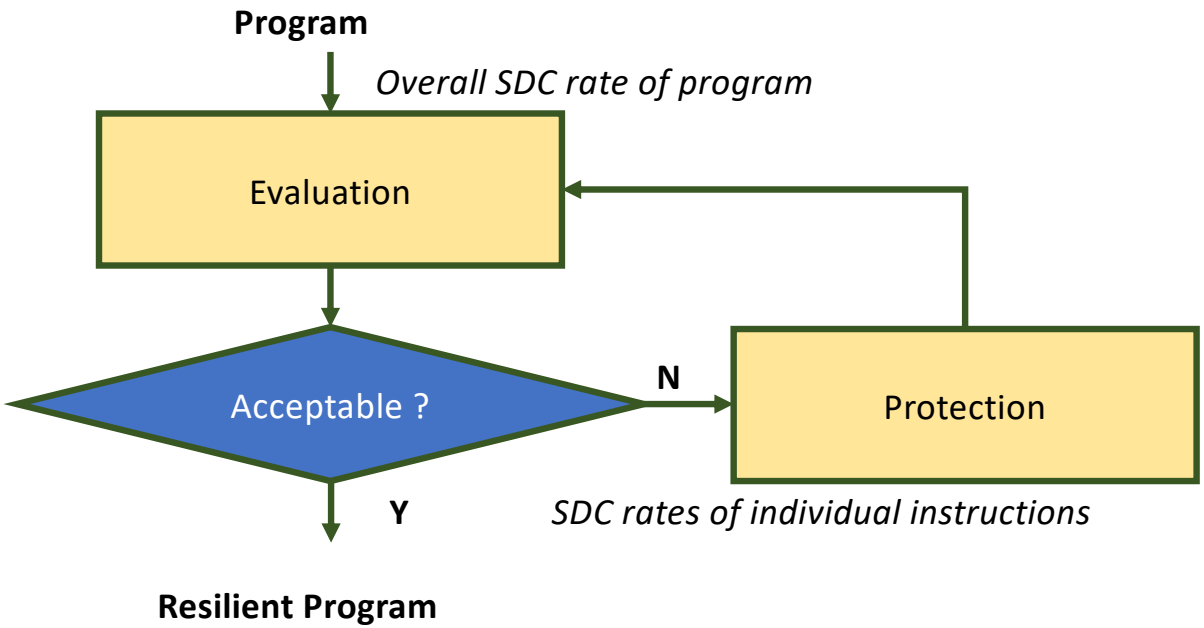
- Motivation and Goals
- Empirical Studies of Error Propagation [DSN'15][ISSRE'15][SC'16]
- Models of Error Propagation [CASES'14] [TECS] [DSN'16] [DSN'18]
- Future Work and Conclusions

SDCTune: Silent Data Corruption (SDCs)



Silent Data Corruption (SDCs): Fault Injection

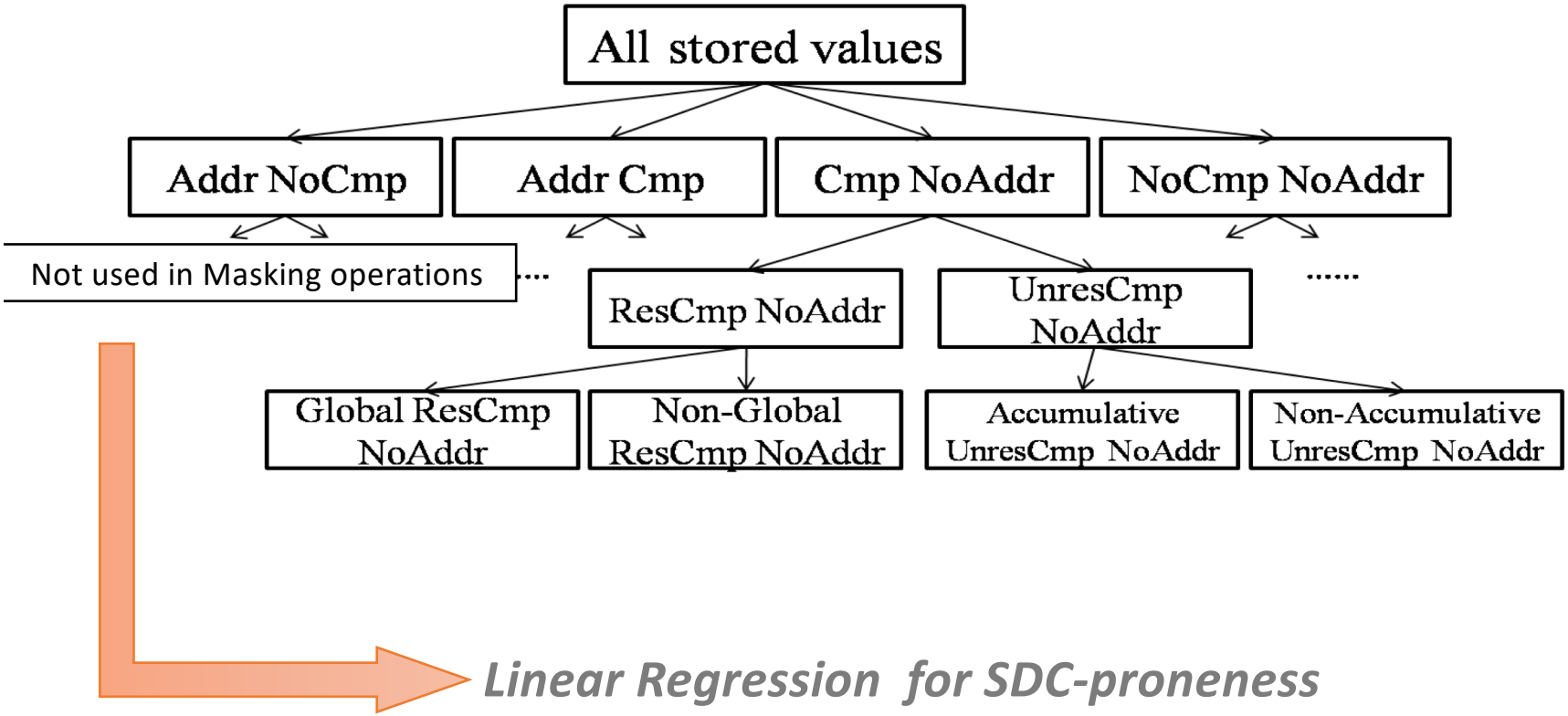
Fault injection approaches take way too long to be practical !



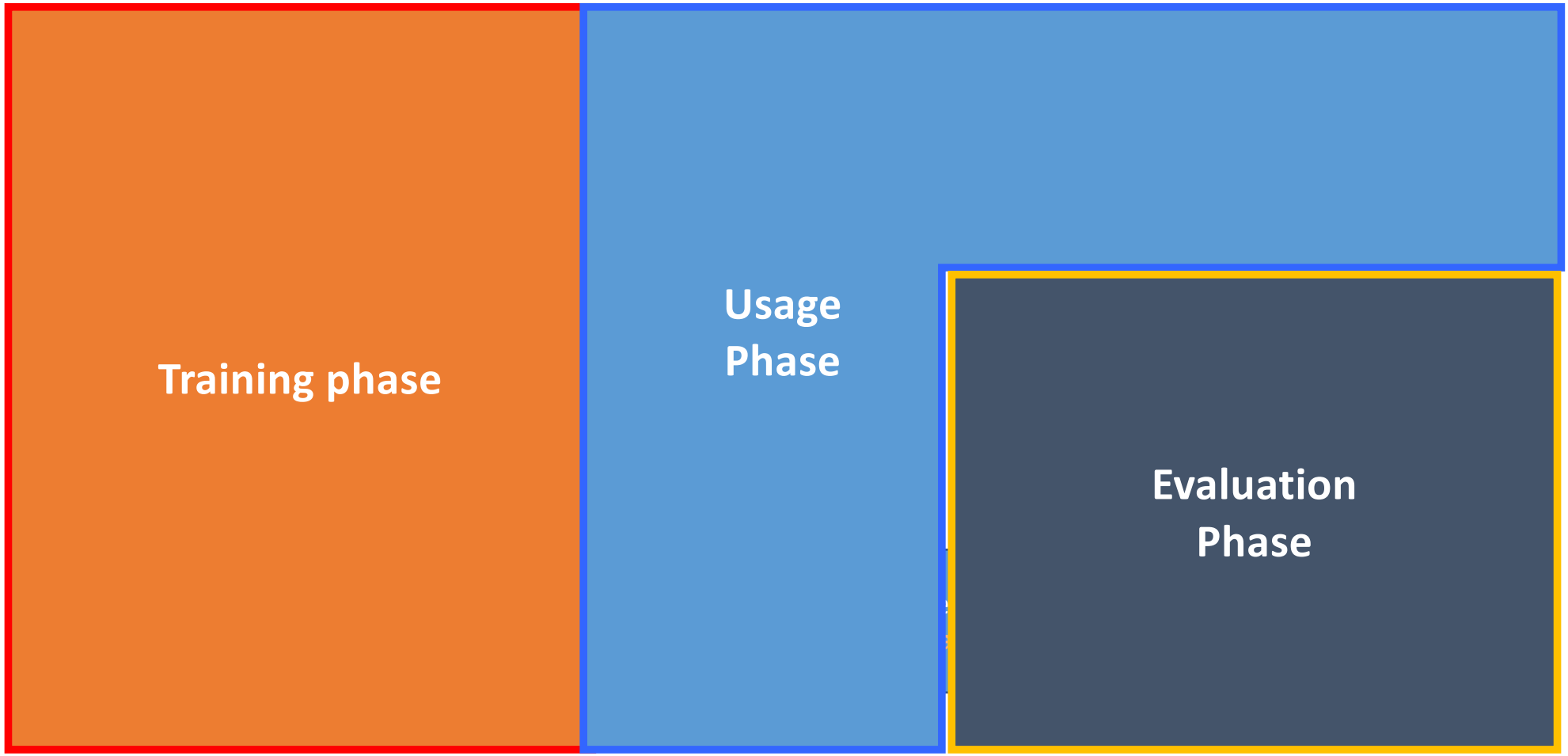
Silent Data Corruption (SDCs):Challenges

- Propagation of SDC is much more complicated
  - Billions of instructions and branches in common dynamic execution
  - Need a comprehensive model
- Our first attempt: SDCTune [CASES'14][TECS]
  - Machine learning algorithm to learn classification and regression tree model

SDCTune: Example Model

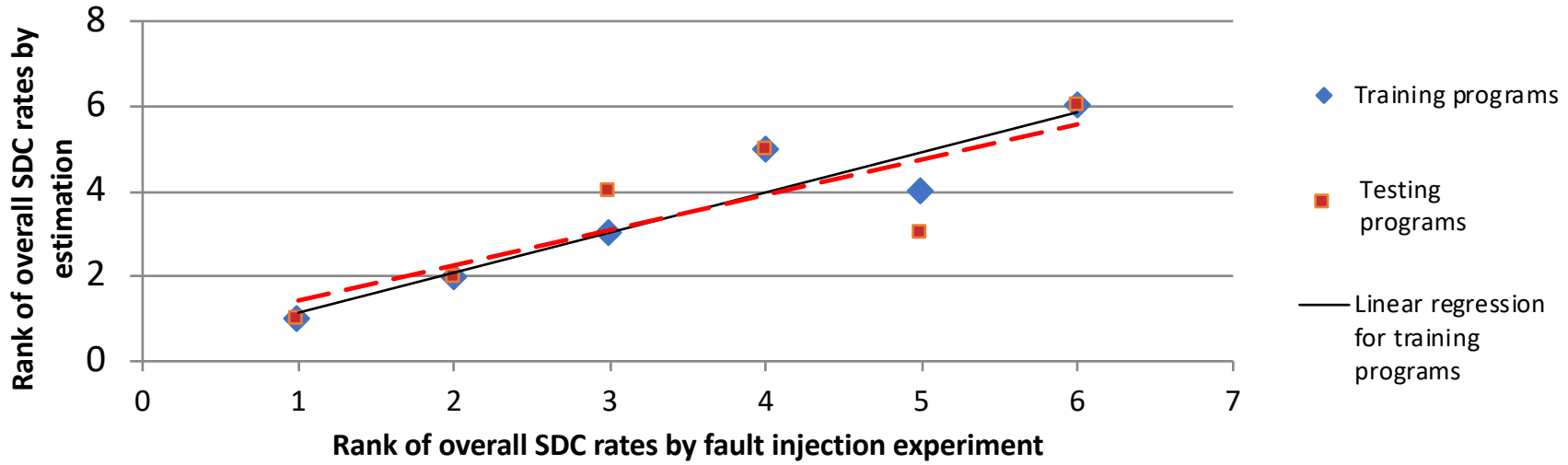


SDCTune: Evaluation Method

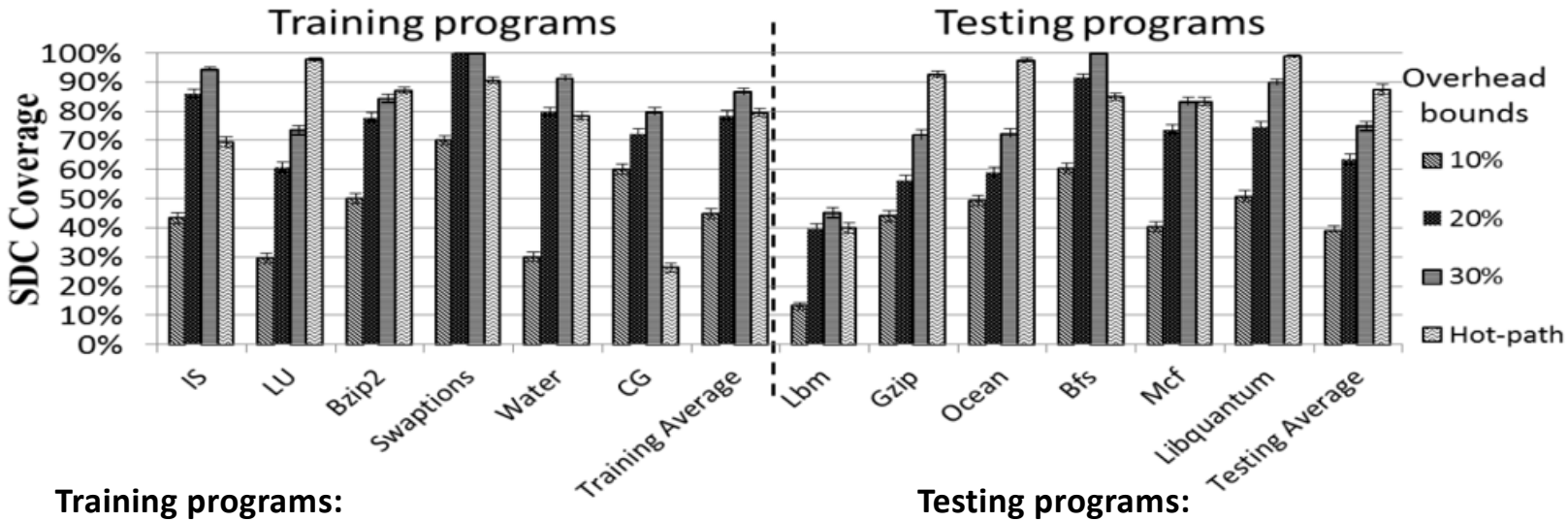


SDCTune: Model Validation

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



SDCTune: SDC Coverage



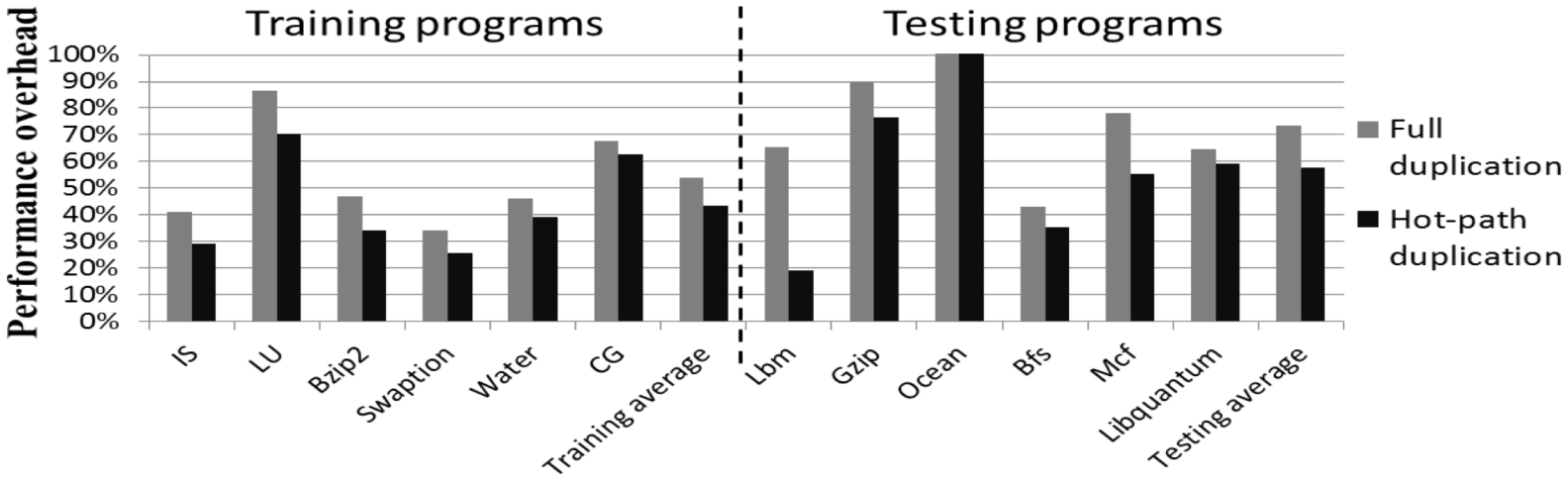
Training programs:

Overhead	Coverage
10%	44.8%
20%	78.6%
30%	86.8%

Testing programs:

Overhead	Coverage
10%	39%
20%	63.7%
30%	74.9%

SDCTune: Full Duplication and Hot-Path Duplication Overheads



Normalized Detection Efficiency	10% overhead	20% overhead	30% overhead
Training programs	2.38	2.09	1.54
Testing programs	2.87	2.34	1.84

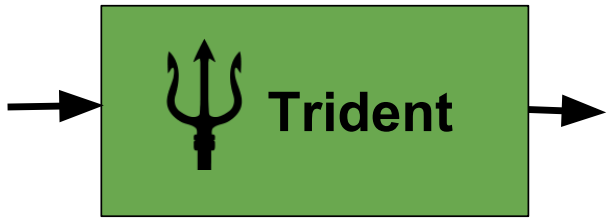
SDCTune: Drawbacks

- Need representative set of benchmarks for training
- Fault injection needs to be performed for every application class
- Little to no explanatory power for analysis
- Cannot be used during early-stage design choice modeling

Trident: Analytical Model

- Error propagations can be decomposed into modules, which can be abstracted into equations without any *fault injections*

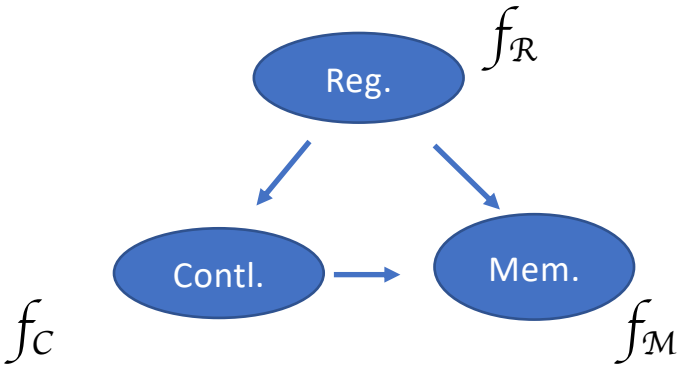
- Program source code (LLVM IR)
- Program input
- Instructions considered as program output

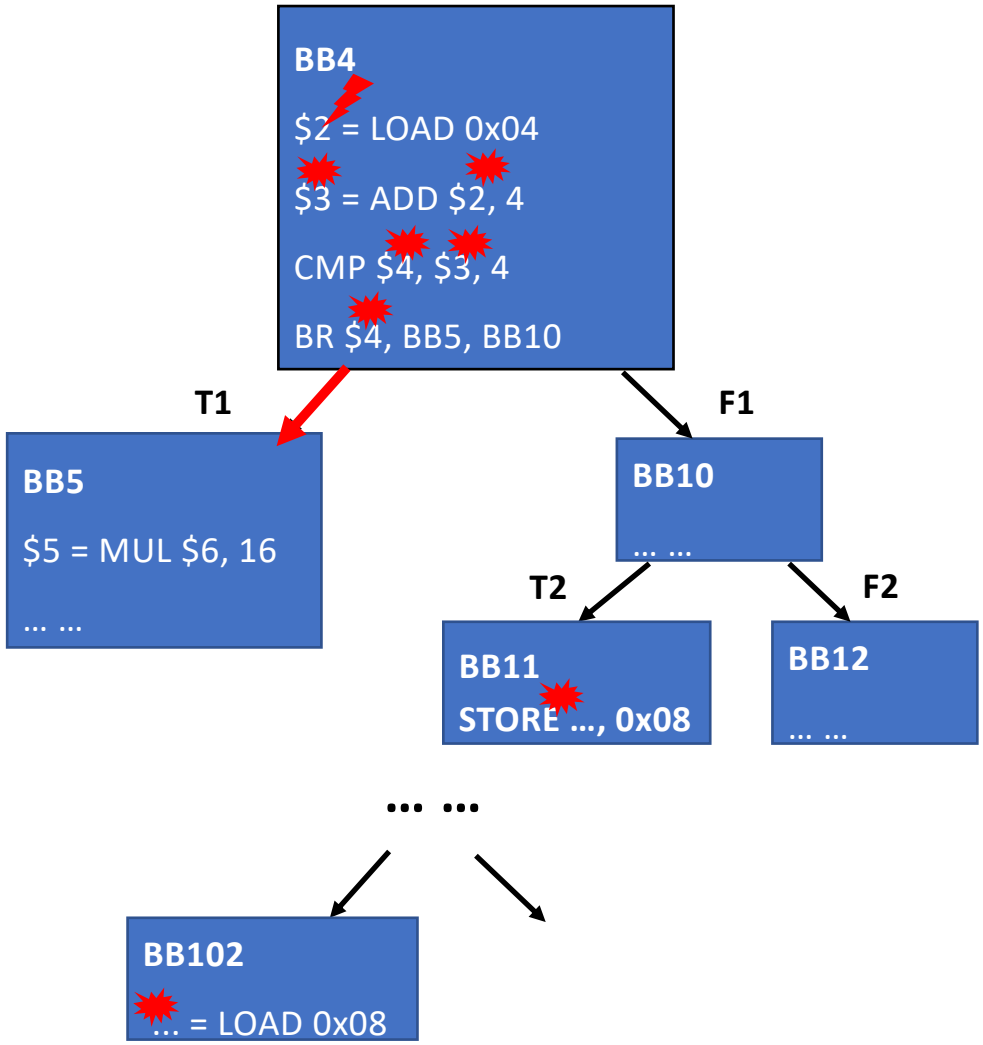


- Overall SDC probability of the program
- SDC probabilities of every instructions

Trident: Approach

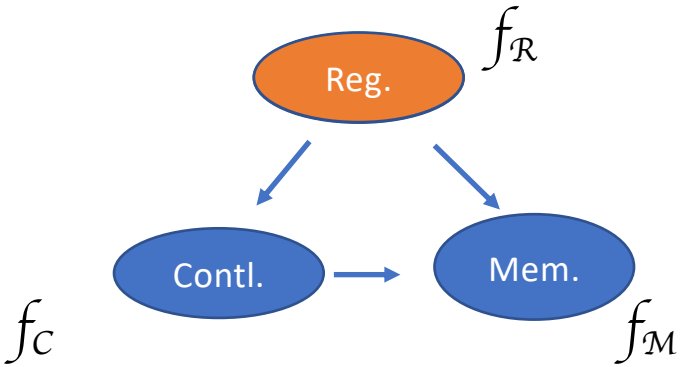
- Three-level modeling
  - Register-communication
  - Control-flow
  - Memory dependency

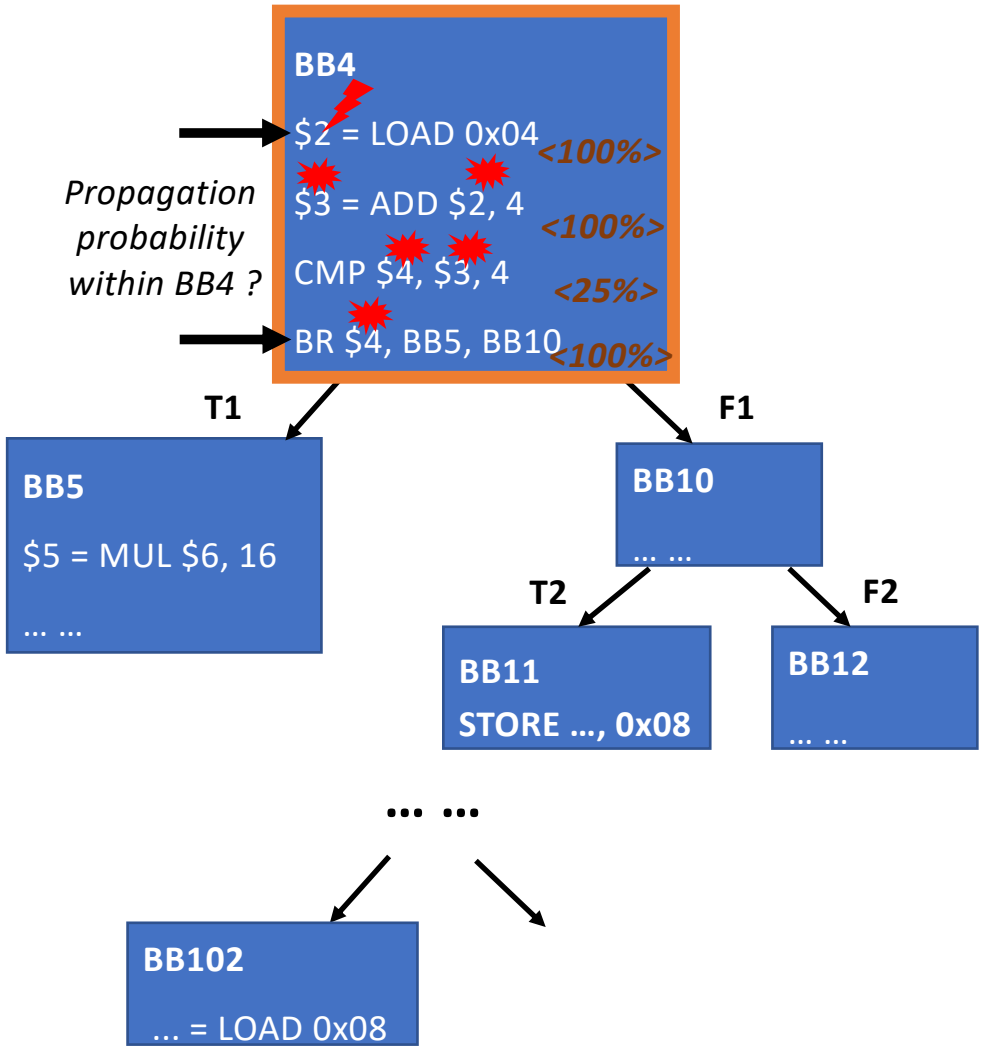




Trident: Approach

$f_{\mathcal{R}}$ = 100% * 100% * 25% * 100% = 25%





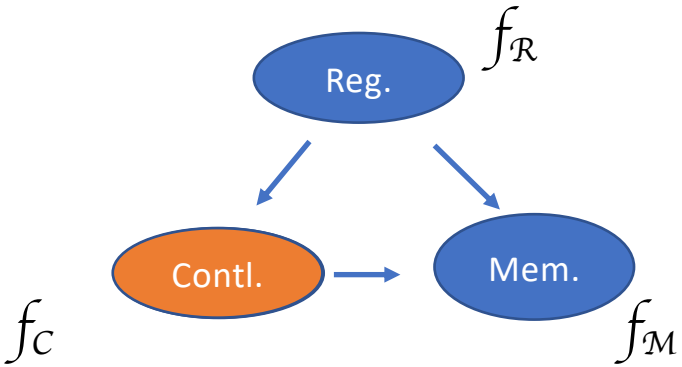
Trident: Approach

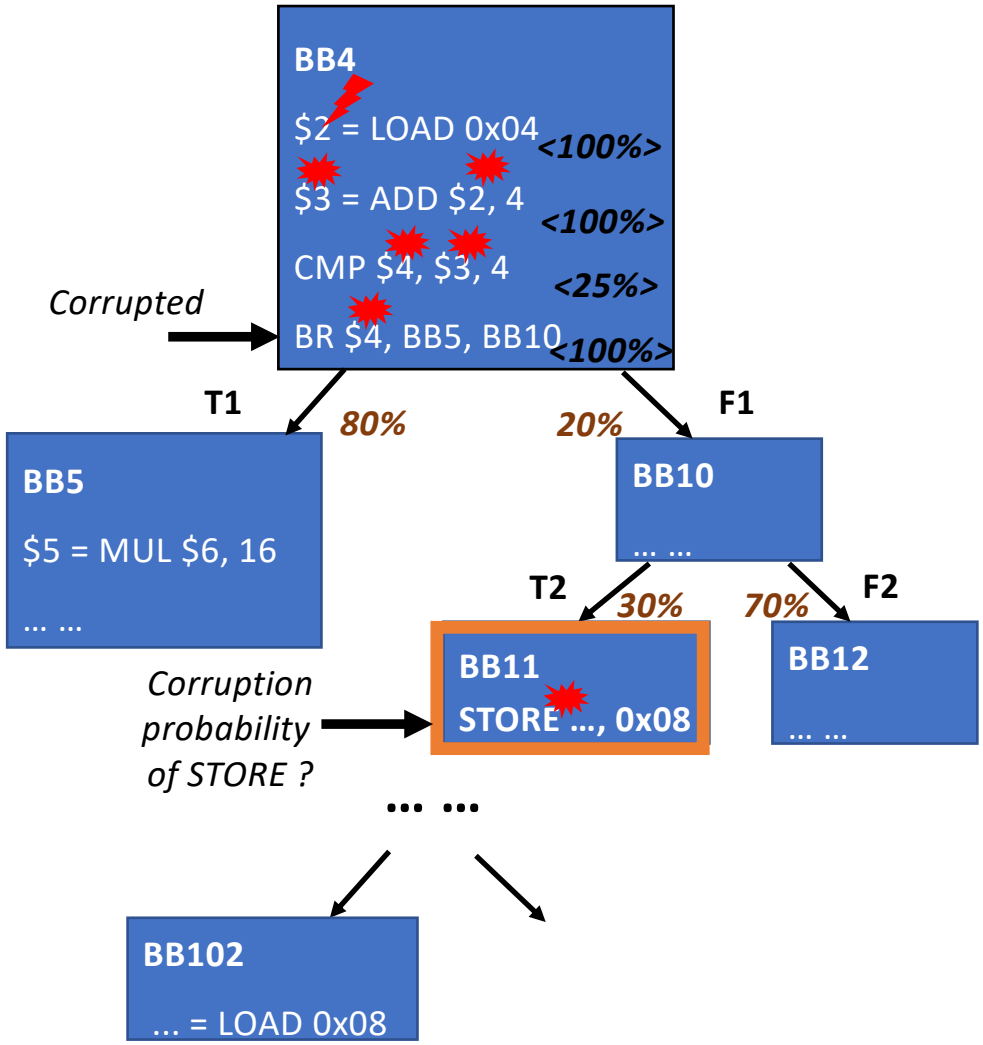
$$f_C = P_e / P_d$$

STORE exec. prob.
*F1*T2*

BR dom. prob.
F1

**For non-loop-terminating branches*

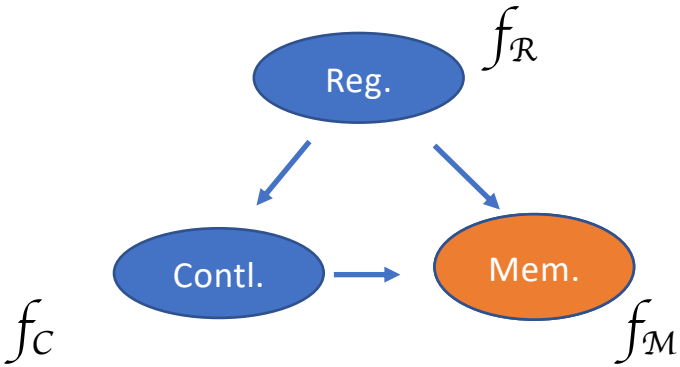


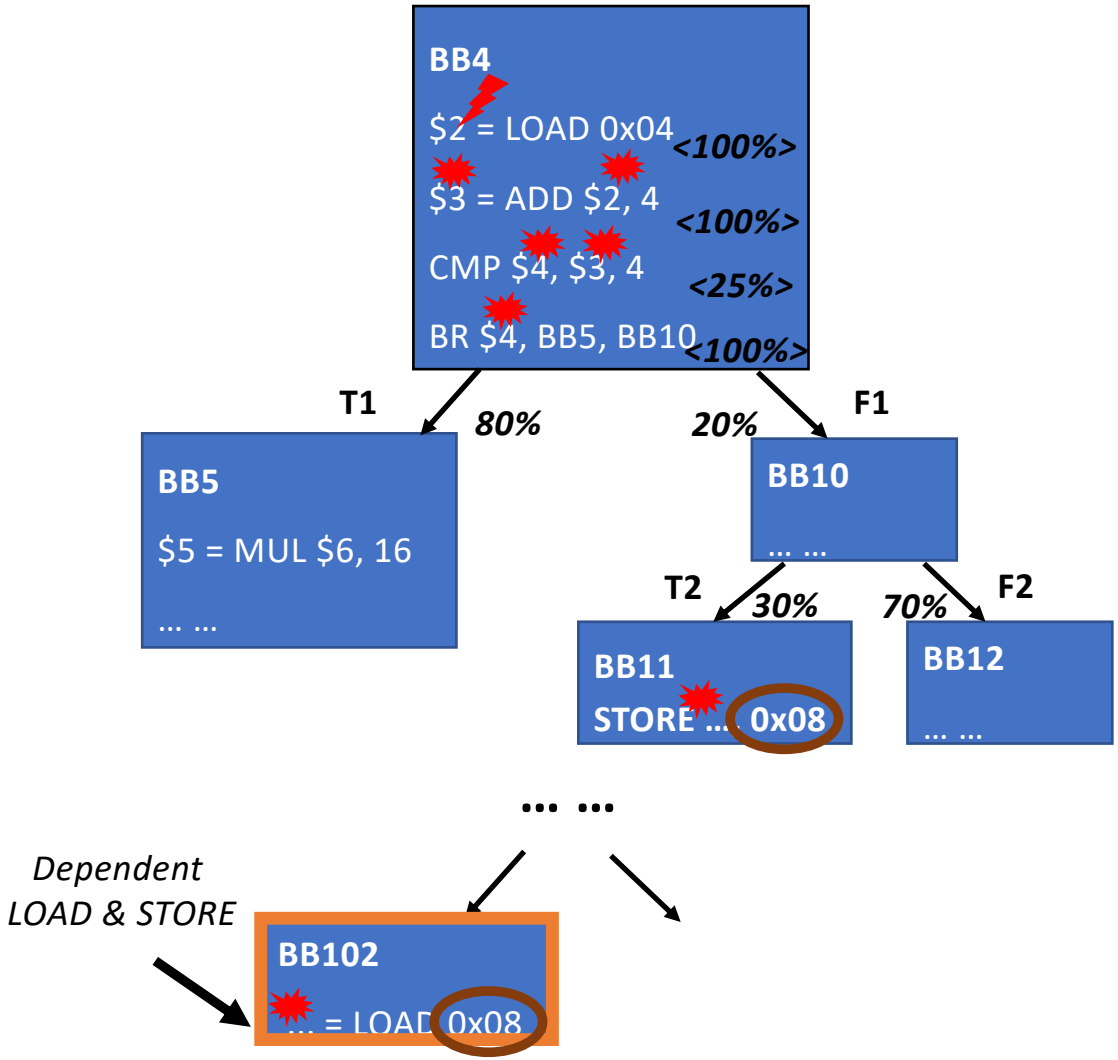


Trident: Approach

$$f_{\mathcal{M}}(\mathcal{I}_n) = f_{\mathcal{R}} * f_C$$

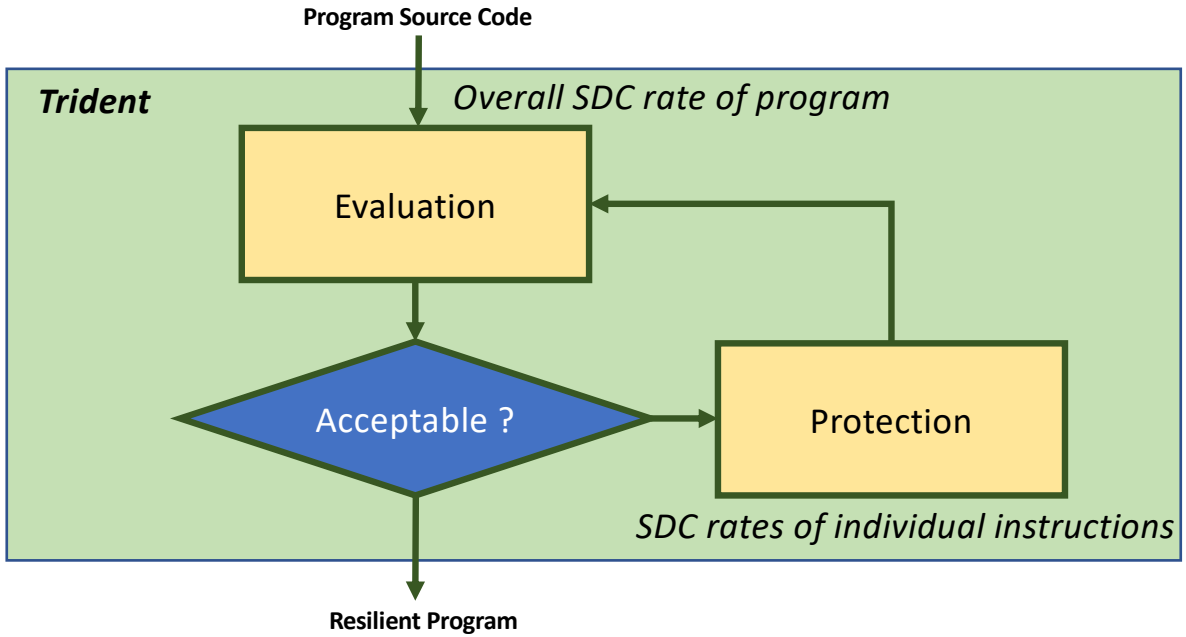
$$\mathcal{P}_{n+1} = f_m(\mathcal{I}_n)$$





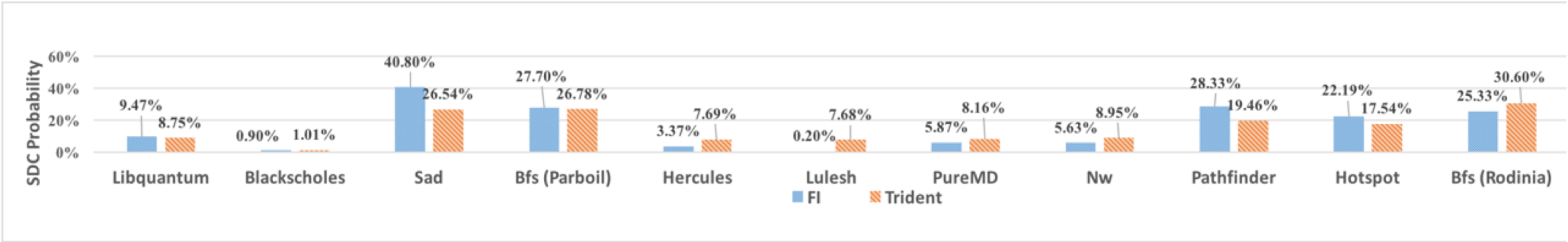
Trident: Implementation

- Built as compiler module
  - Integrated with LLVM/Clang
  - Validated with 11 benchmarks
- Online tuning for resilience
  - Extend to GPU programs



Trident: Results

Error Bar: +/-0.07% ~ +/-1.76% at 95% Confidence Interval

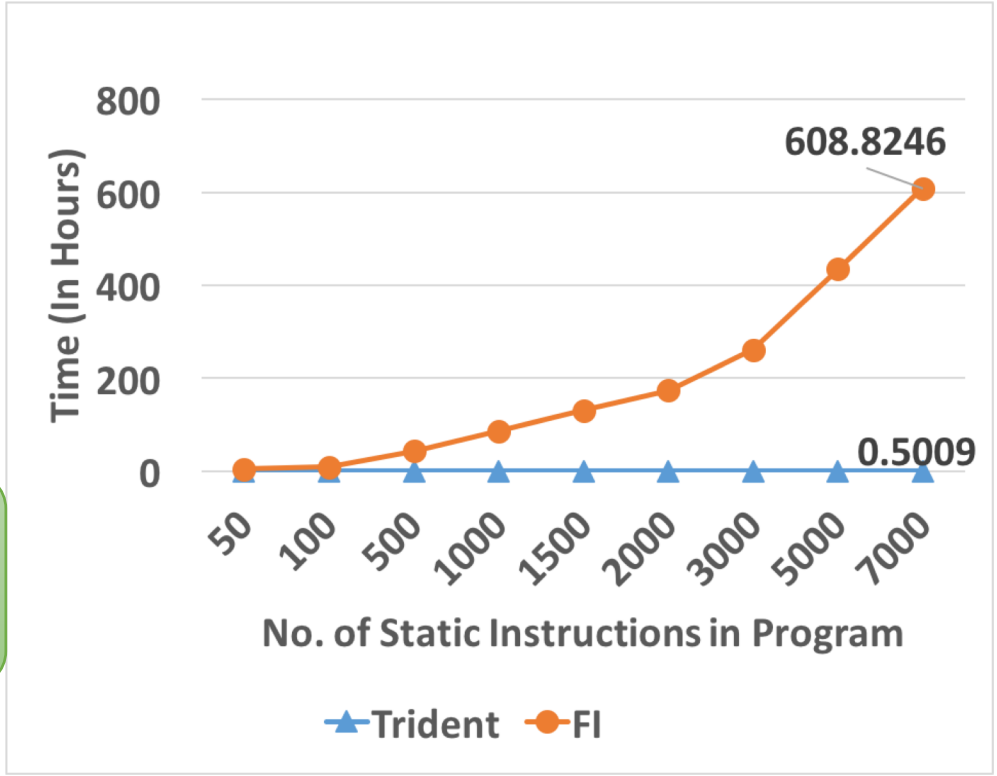


- Mean Absolute Error: 4.75%
- Much better than simpler models
- T-test to compare with fault injection

Trident: Performance Speedup

- **Average speedup**
  - Fault Injection: ~96 hrs
  - Trident: ~15 mins

More than 2 orders of magnitude faster



vTrident: Input-Dependent Error Propagation

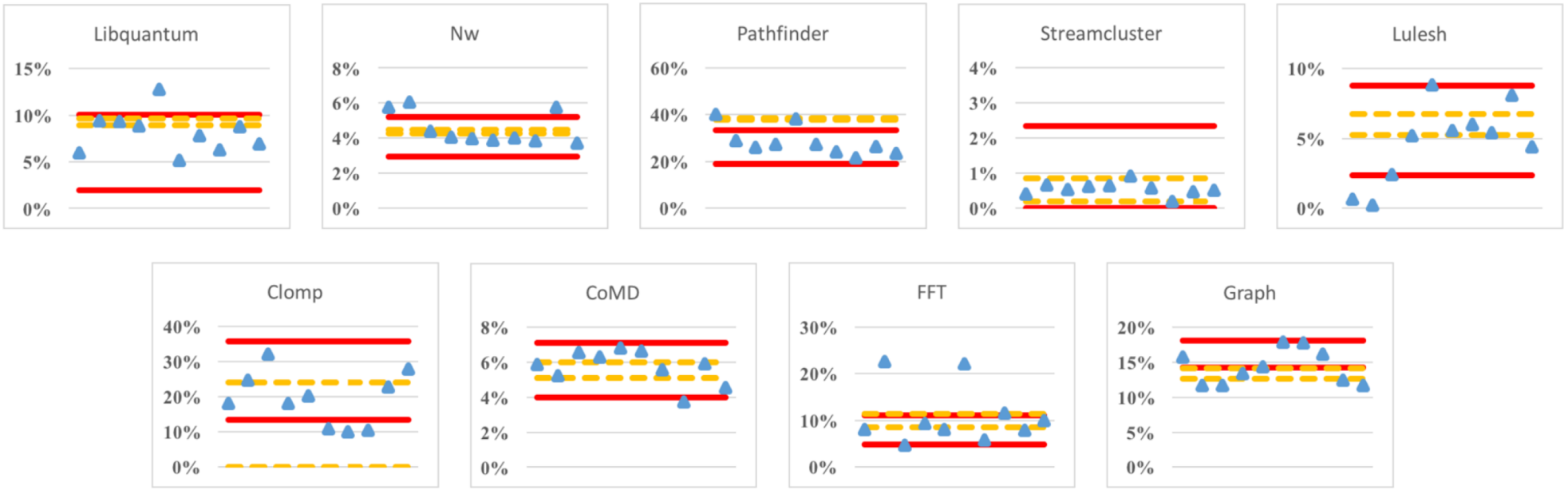
- A program is executed with multiple inputs in production
  - SDC probabilities highly depend on inputs
  - Need to understand how inputs affect error propagation
  - Need to bound SDC probabilities across multiple inputs
- Two kinds of volatilities decide overall SDC probability :

$$P_{overall} = N_{SDC}/N_{total} \quad (1)$$

$$= (\sum_{i=1}^{n} P_i * N_i)/N_{total} = \sum_{i=1}^{n} P_i * (N_i/N_{total}) \quad (2)$$

Instruction-SDC-Volatility ***Instruction-Execution-Volatility***

vTrident Results: Bounding SDC Rates



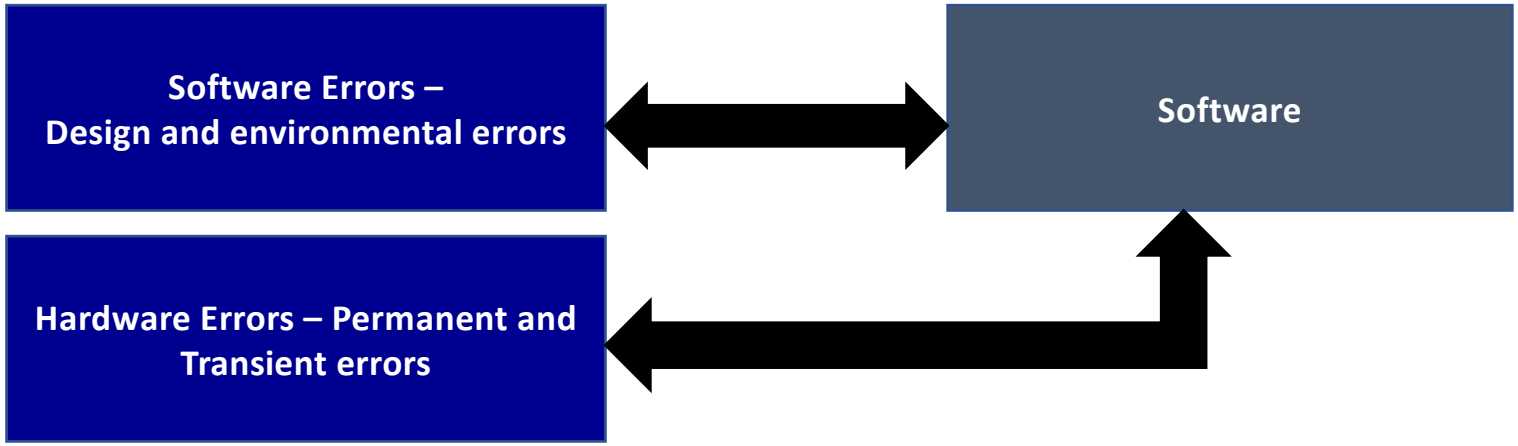
About 80% of the measured SDC values are within the bounds identified by vTrident

Outline

- Motivation and Goals
- Empirical Studies of Error Propagation [DSN'15] [ISSRE'15][SC'16]
- Models of Error Propagation [CASES'14] [TECS] [DSN'16] [DSN'18]
- Future Work and Conclusions

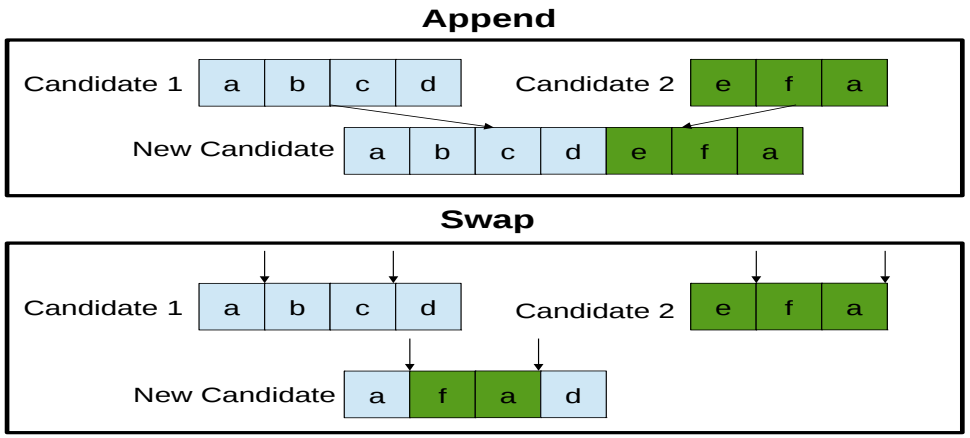
Future Work: Formal Reasoning

- **Need to verify programs in presence of faults**
  - Consider both hardware and software faults
  - Model-checking to find corner cases [DSN'08][TC]
  - Integration with software development process



Future Work: Genetic Programming

- Evolve applications using Genetic Programming (GP)
- Preliminary study on finding “resilience friendly” compiler optimizations
- Can improve both resilience and performance simultaneously [EDCC’16]

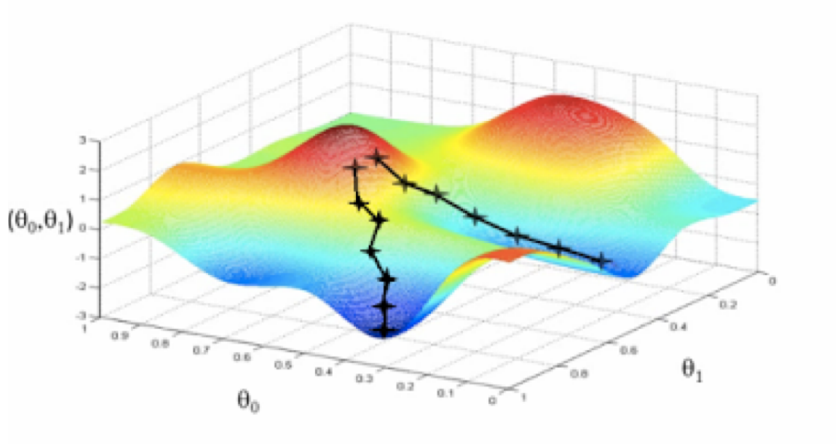


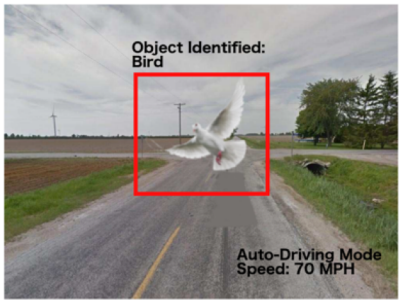
Future Work: Resilient ML

Deriving ML algorithms resilient to perturbations

- Small changes → Similar outputs
- Convergence properties in presence of faults

Preliminary study of DNN applications - Found catastrophic cases [SC 2017]





Conclusion

- **Error Propagation is the reason for most catastrophic failures in systems**
  - Need systematic approaches to identify and mitigate error propagation
- **Empirical approach**
  - Apply heuristics and ML to identify error propagation
- **Analytical model**
  - Decomposition and abstraction
  - Almost as accurate as fault injections for fraction of cost

Acknowledgements

(Website: http://blogs.ubc.ca/karthik)