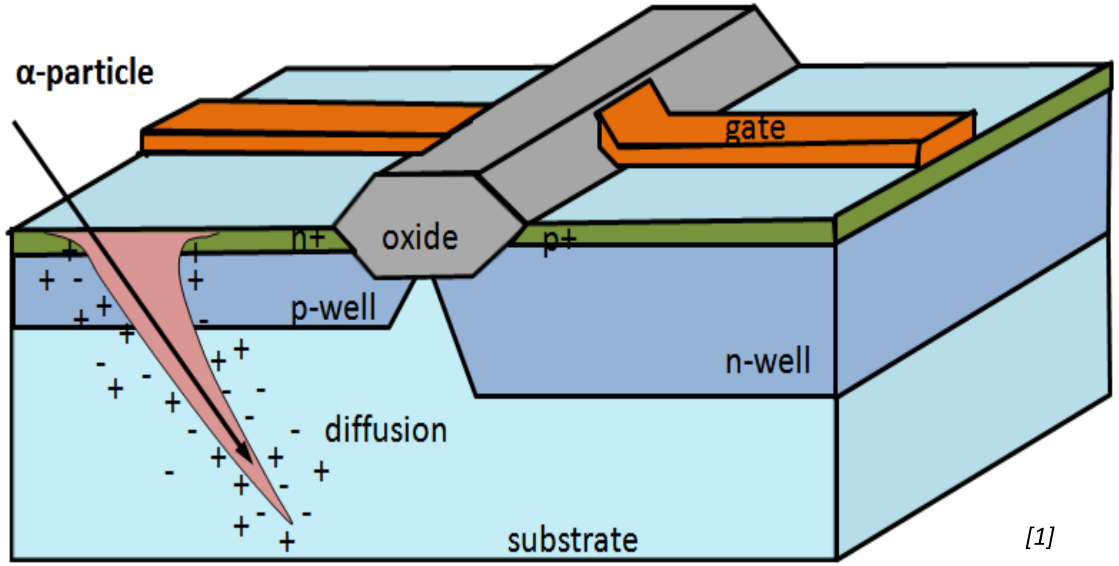
Modeling Soft-Error Propagation in Programs

Guanpeng (Justin) Li
Karthik Pattabiraman

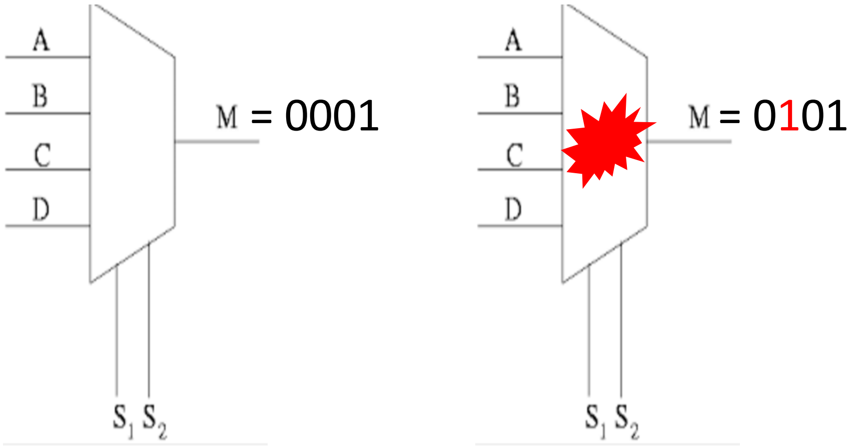
Siva Hari
Michael Sullivan
Timothy Tsai

Motivation: Soft Errors



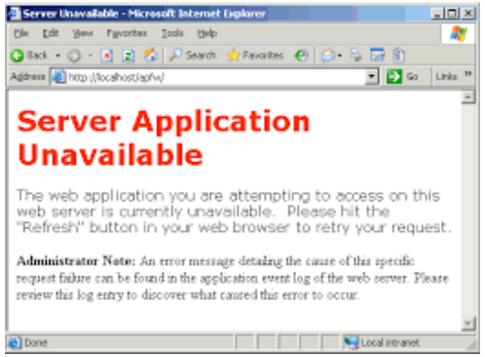
[1]

Soft errors becoming more common in processors



[1] http://aviral.lab.asu.edu/soft-error-resilience/

Silent Data Corruption (SDC)

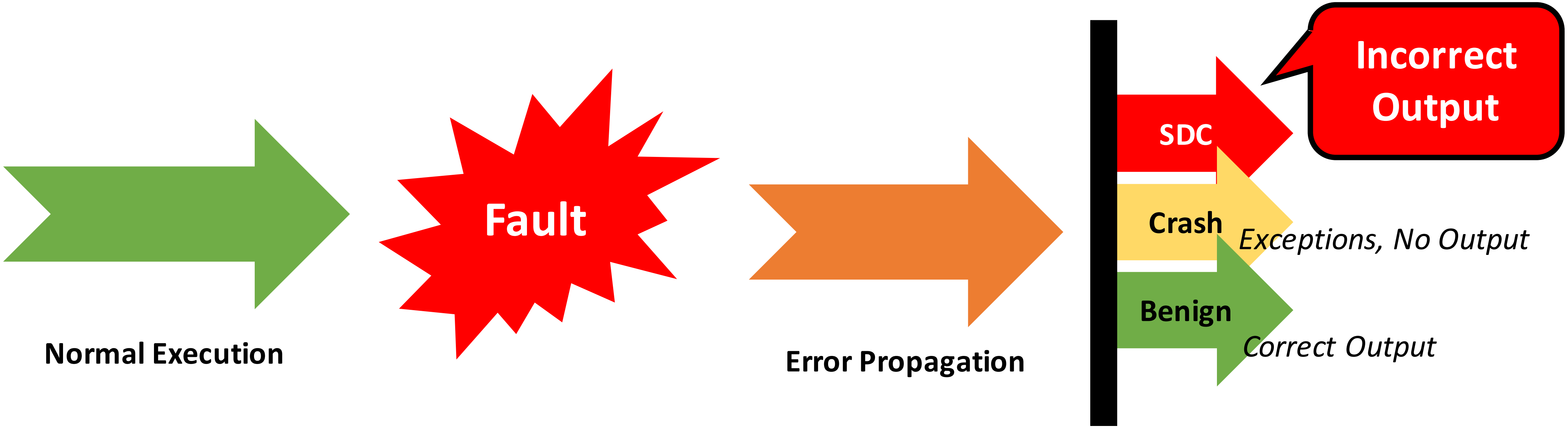

Server Unavailable - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/apfw/
Go Links
Done
Local intranet

Server Application Unavailable

The web application you are attempting to access on this web server is currently unavailable. Please hit the "Refresh" button in your web browser to retry your request.

Administrator Note: An error message detailing the cause of this specific request failure can be found in the application event log of the web server. Please review this log entry to discover what caused this error to occur.

Amazon S3 Incident



Fault

Normal Execution

Error Propagation

SDC

Incorrect Output

Crash

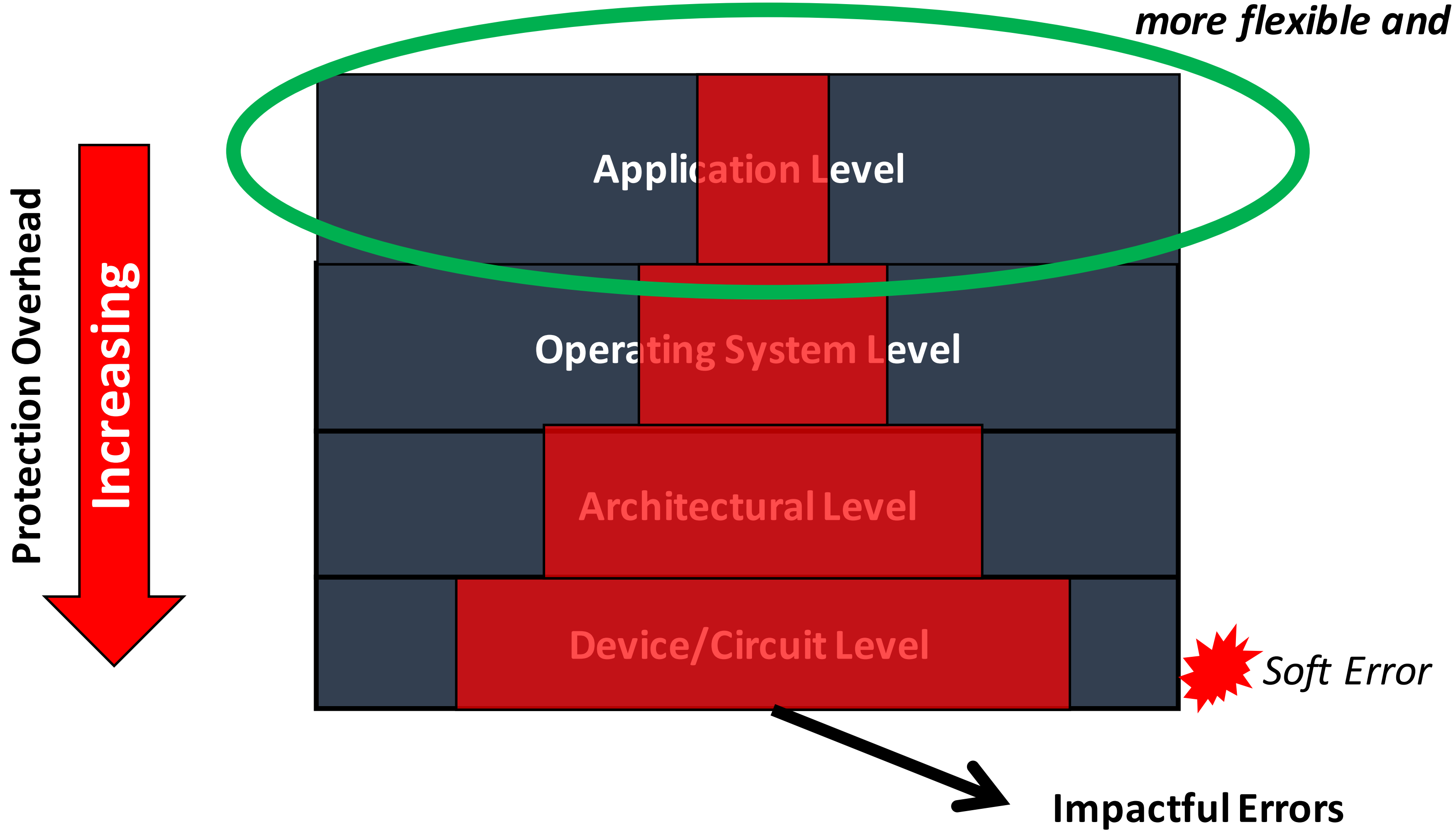
Exceptions, No Output

Benign

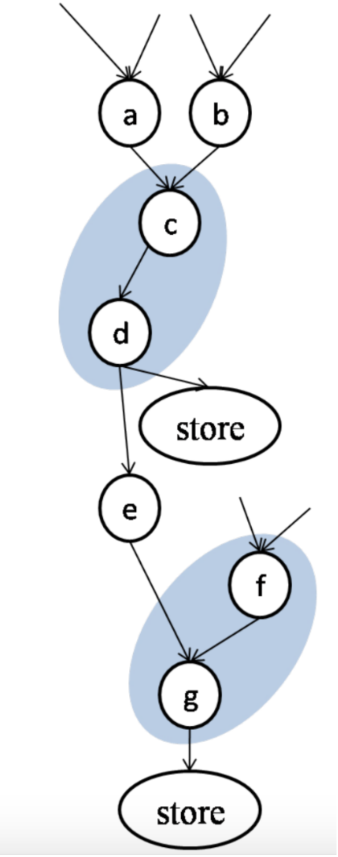
Correct Output

Software Solutions

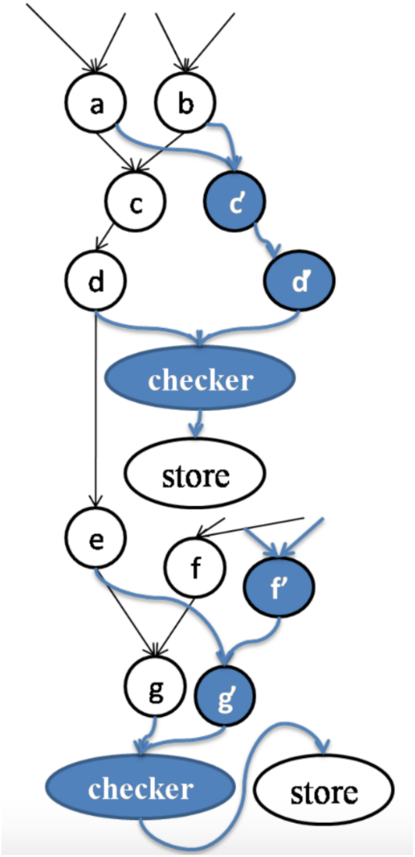
Software protection techniques are more flexible and cost-effective!



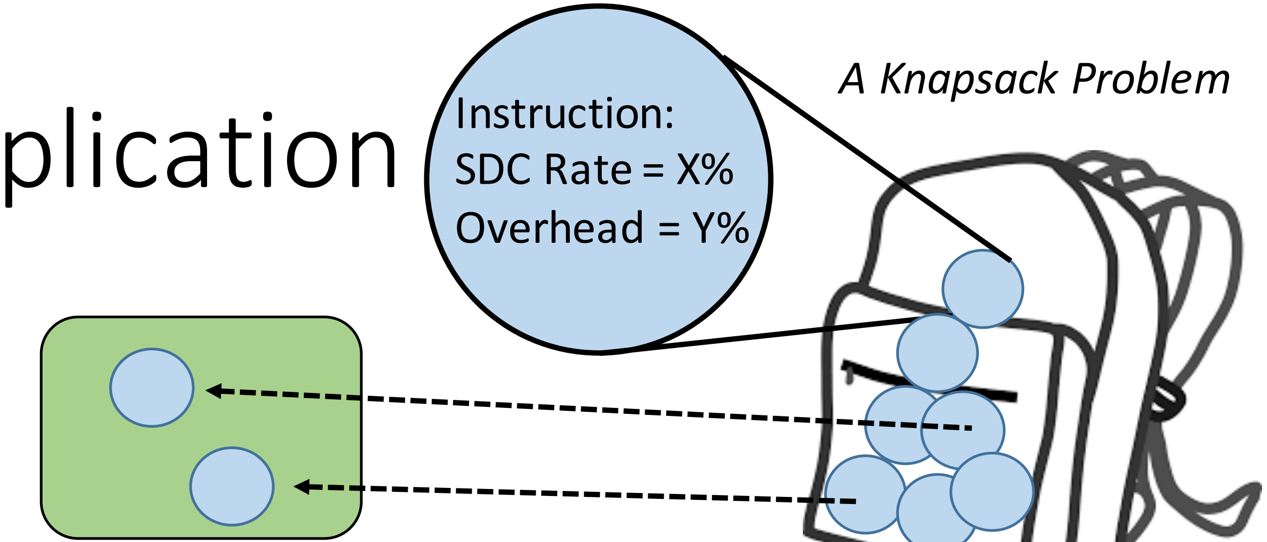
Selective Instruction Duplication



Instruction Sequence



Instruction Duplication

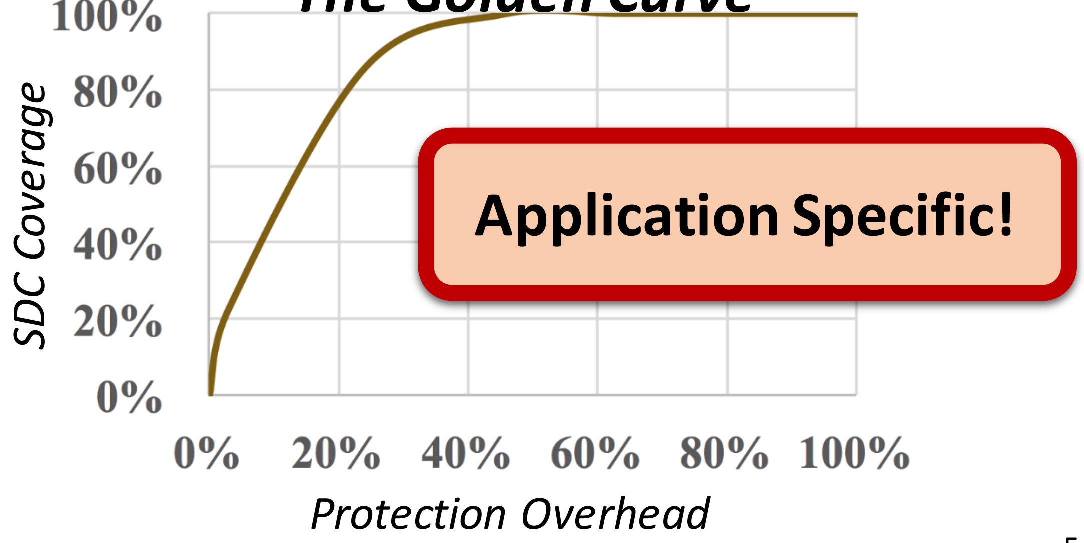


A Knapsack Problem

Instruction:
SDC Rate = X%
Overhead = Y%

Selected Instructions for Given Target SDC Coverage

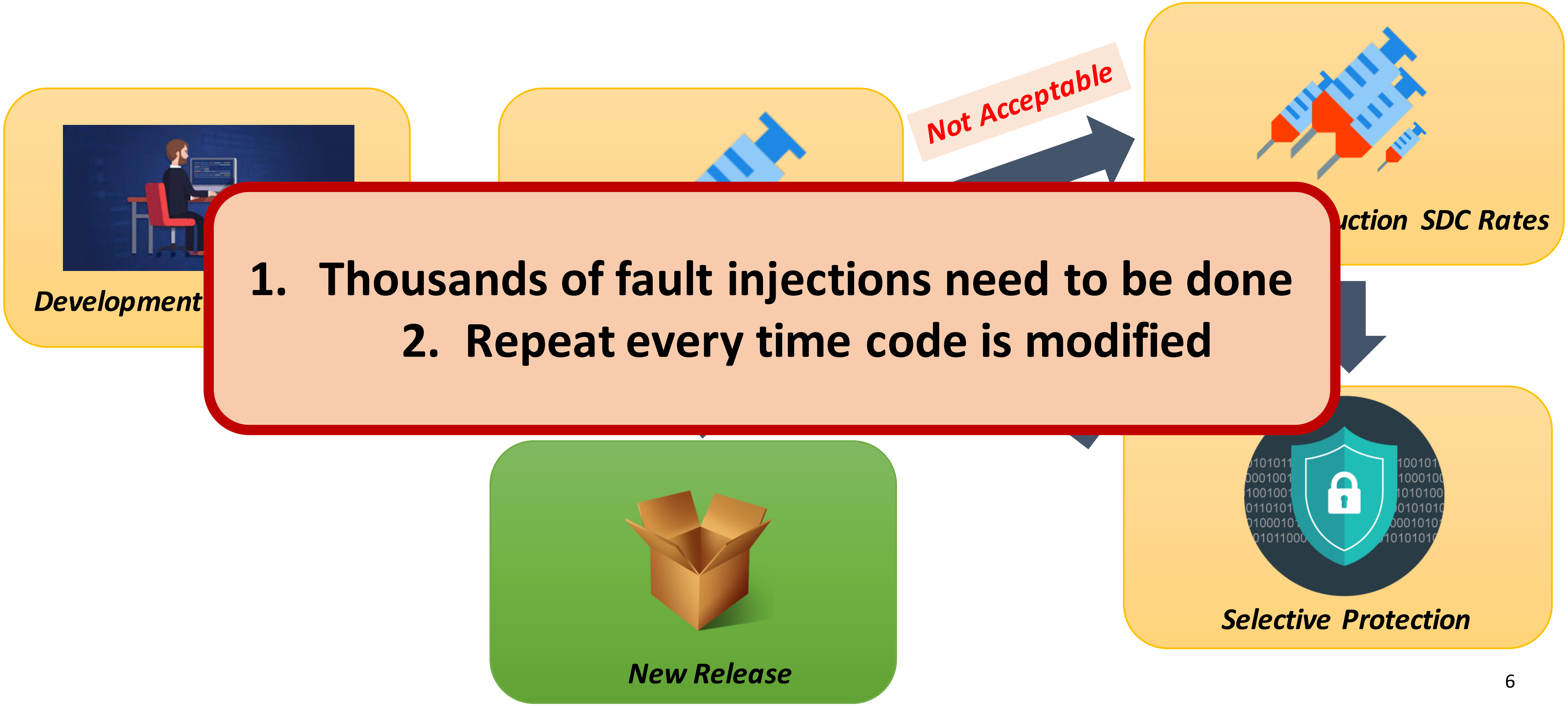
"The Golden Curve"



Application Specific!

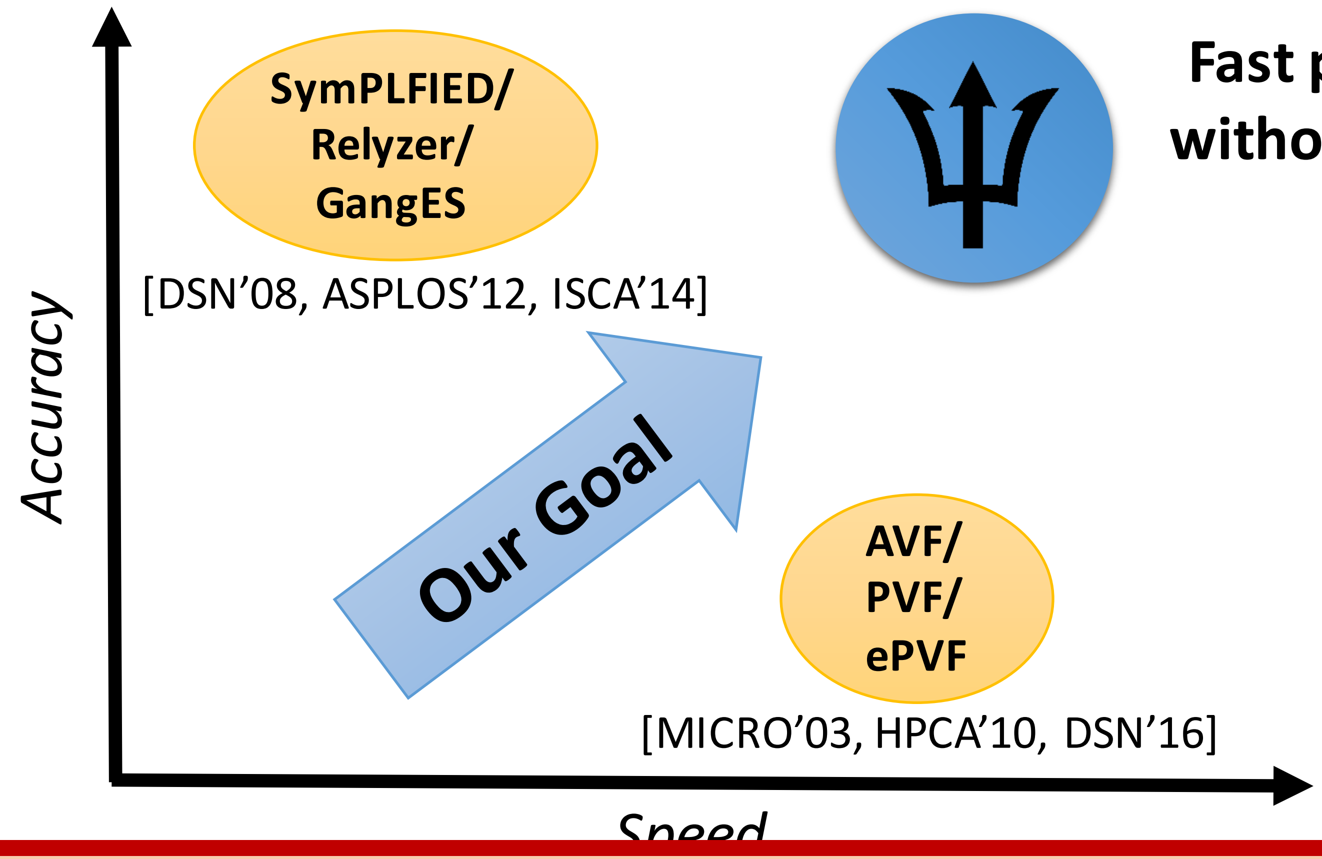
**Measured in Libquantum, SPEC*

Developing Fault-Tolerant Applications



1. **Thousands of fault injections need to be done**
2. **Repeat every time code is modified**

Our Goal

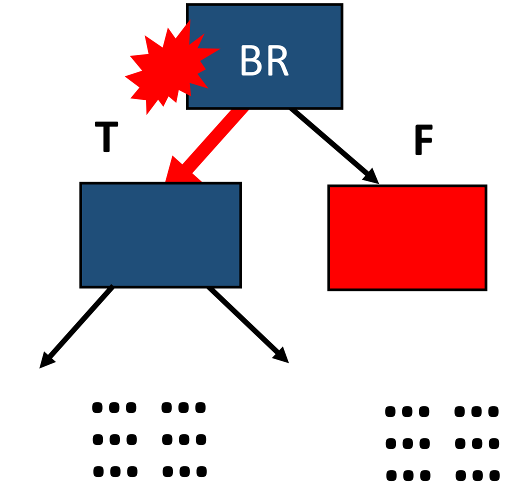


Fast prediction of SDC without fault injection!

No existing technique models error propagation in both fast and accurate way!

Challenges

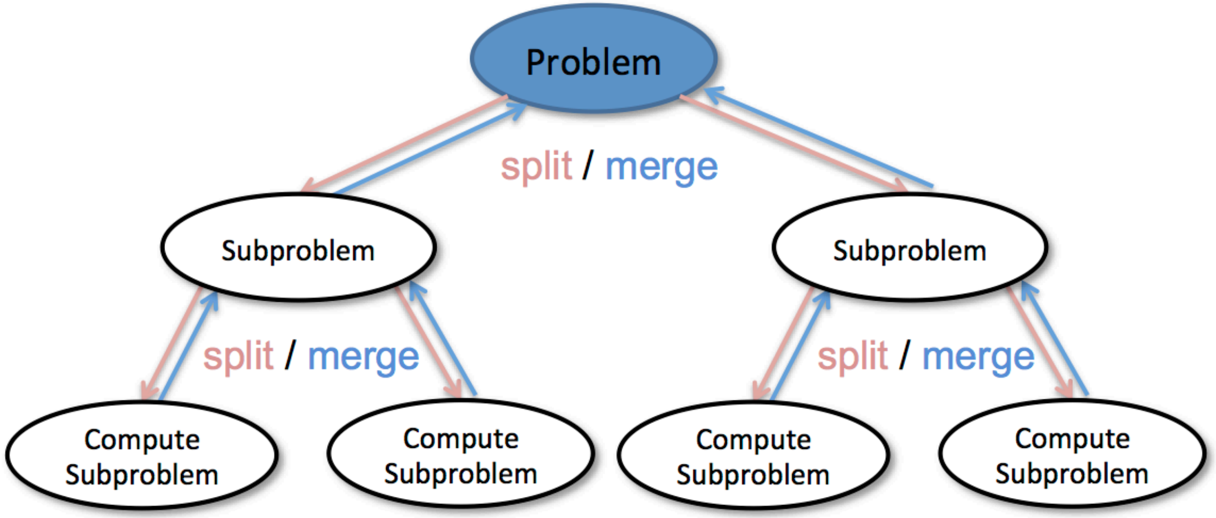
- Tracking SDC propagation is hard
  - Over billions of executed instructions
  - Every instruction may propagate errors with different probabilities
- Dynamic nature of program execution
  - Control-flow divergence



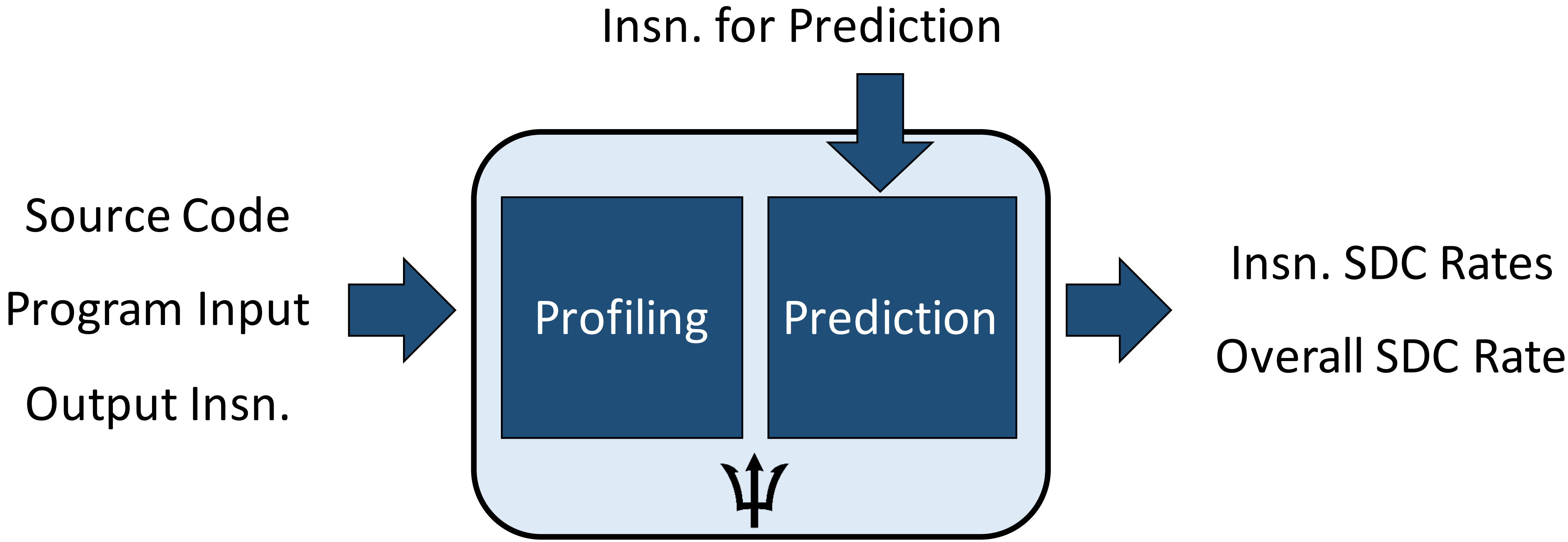
Corrupting subsequent states

Trident: Key Insight

- Error propagations can be decomposed into modules, which can be abstracted into probabilistic events
  - Decomposition
  - Abstraction



Trident: Workflow



Insn. for Prediction

Source Code

Program Input

Output Insn.

Profiling

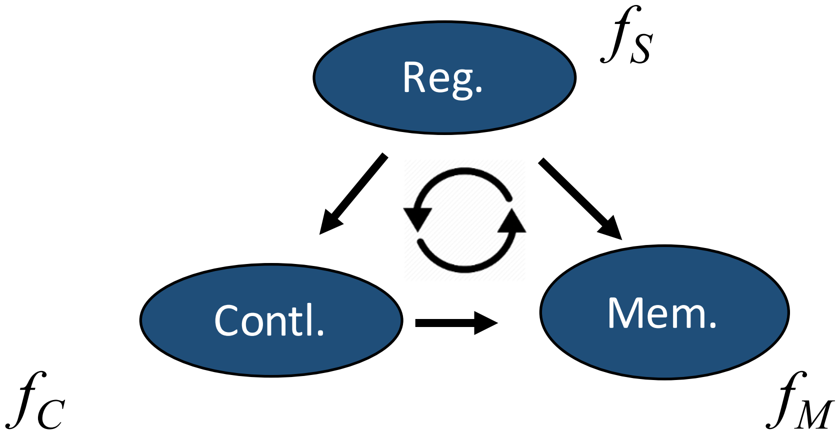
Prediction

Insn. SDC Rates

Overall SDC Rate

Trident: Our Approach

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



Reg.

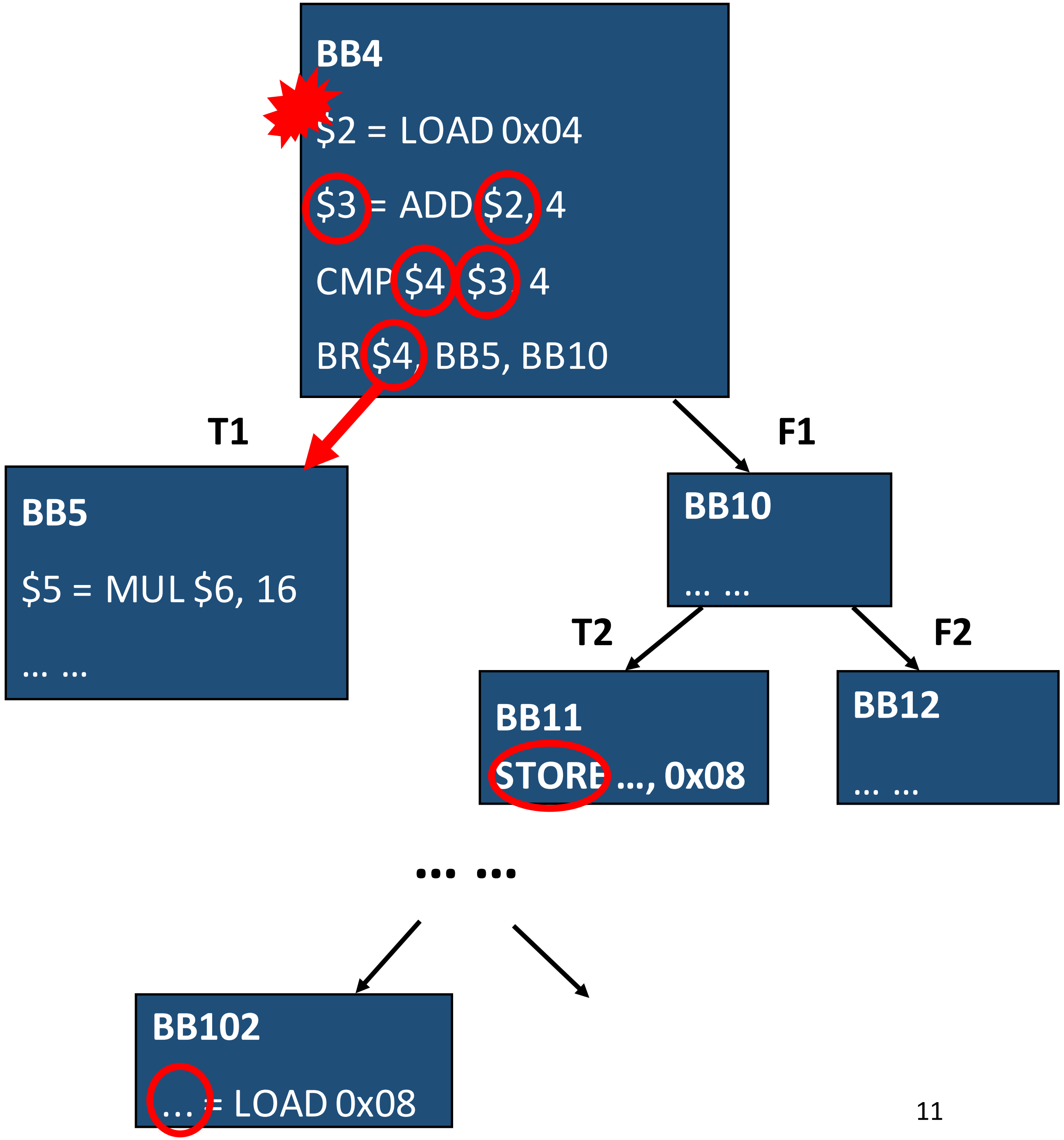
f_S

Contl.

f_C

Mem.

f_M



BB4

```
$2 = LOAD 0x04
$3 = ADD $2, 4
CMP $4, $3, 4
BR $4, BB5, BB10
```

T1

F1

BB5

```
$5 = MUL $6, 16
… …
```

BB10

… …

T2

F2

BB11

```
STORE …, 0x08
```

BB12

… …

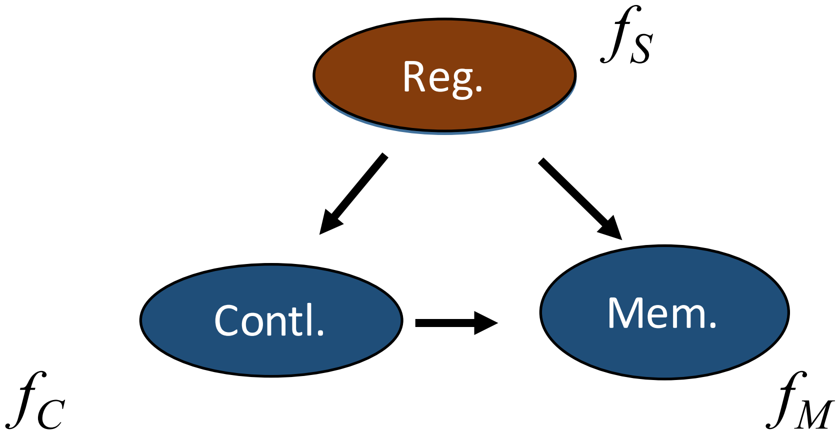
… …

BB102

```
… = LOAD 0x08
```

Trident: Register Commn.

f_s = *100% * 100% * 25% * 100% = 25%*



f_S

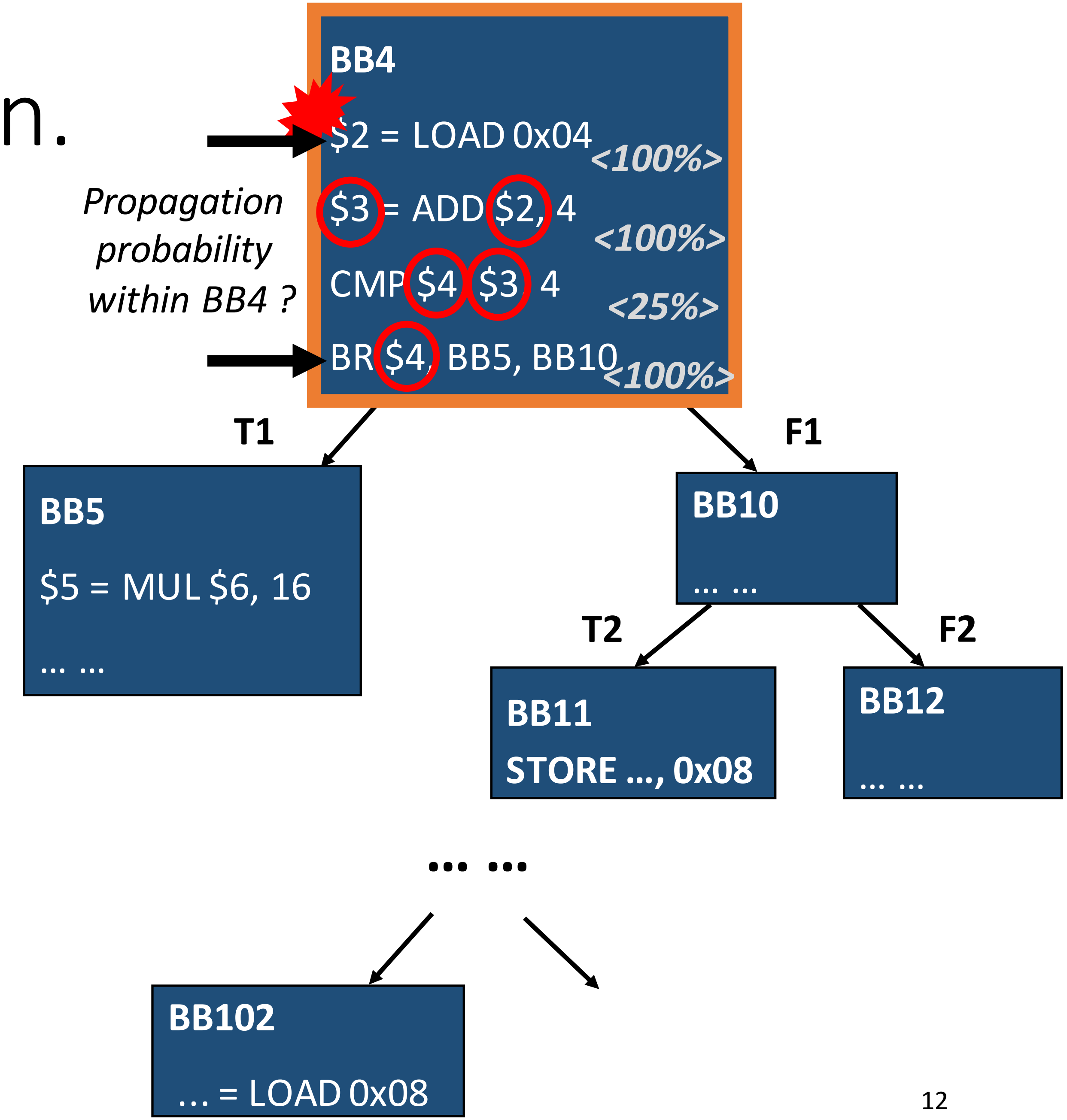
Reg.

Contl.

Mem.

f_C

f_M



Propagation probability within BB4 ?

BB4

$2 = LOAD 0x04 *<100%>*

$3 = ADD $2, 4 *<100%>*

CMP $4 $3, 4 *<25%>*

BR $4, BB5, BB10 *<100%>*

T1

F1

BB5

$5 = MUL $6, 16

… …

BB10

… …

T2

F2

BB11

STORE …, 0x08

BB12

… …

… …

BB102

… = LOAD 0x08

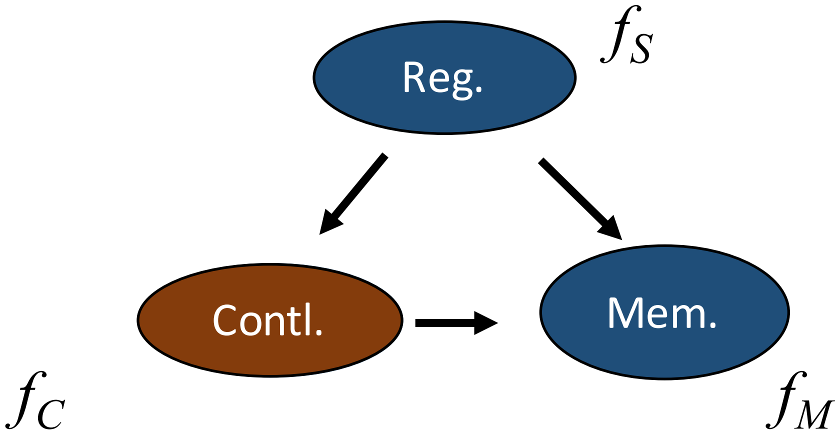
Trident: Control-Flow

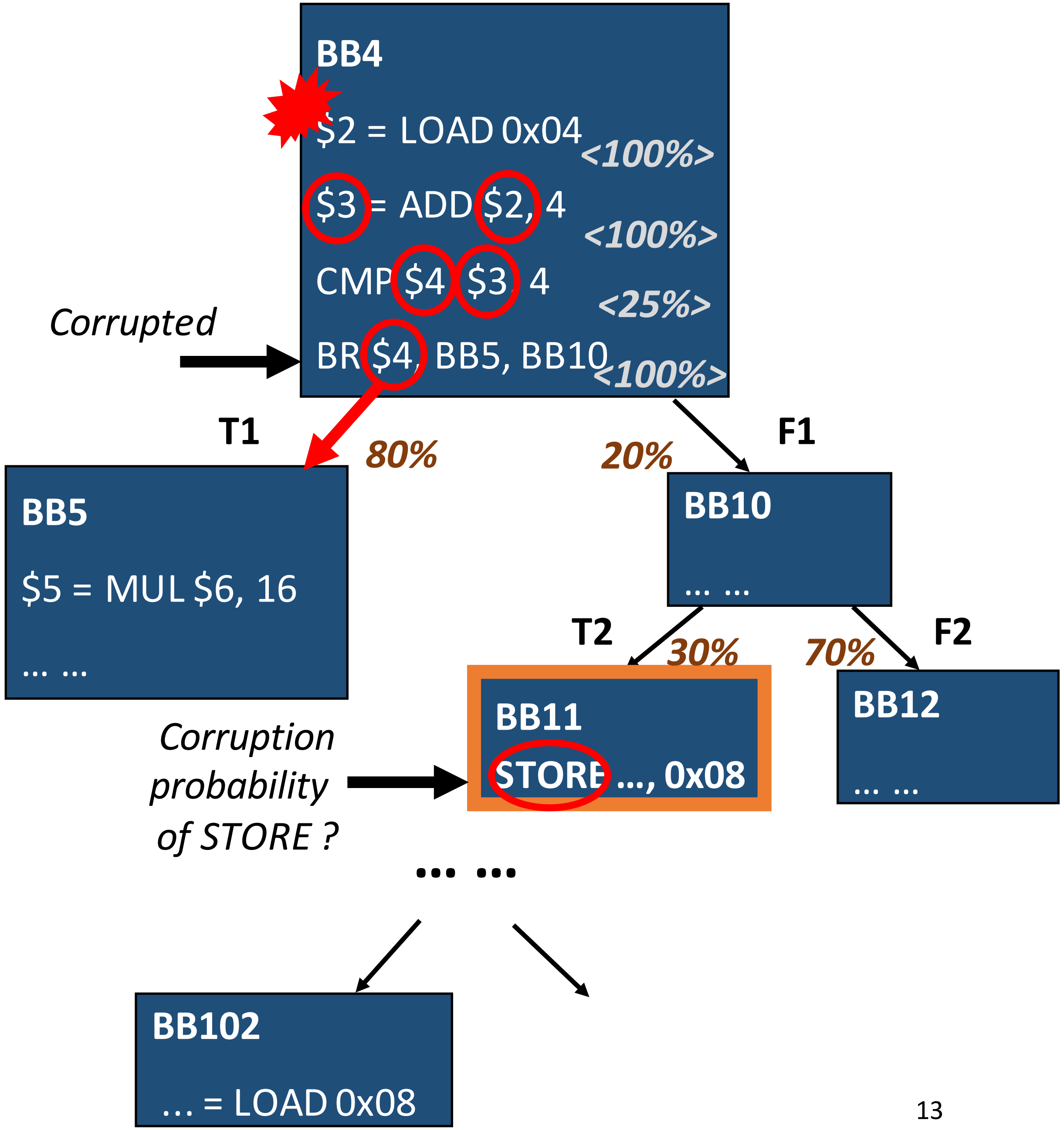
$$f_C = P_e / P_d$$

STORE exec. prob.
*F1*T2*

BR dom. prob.
F1

**For non-loop-terminating branches*

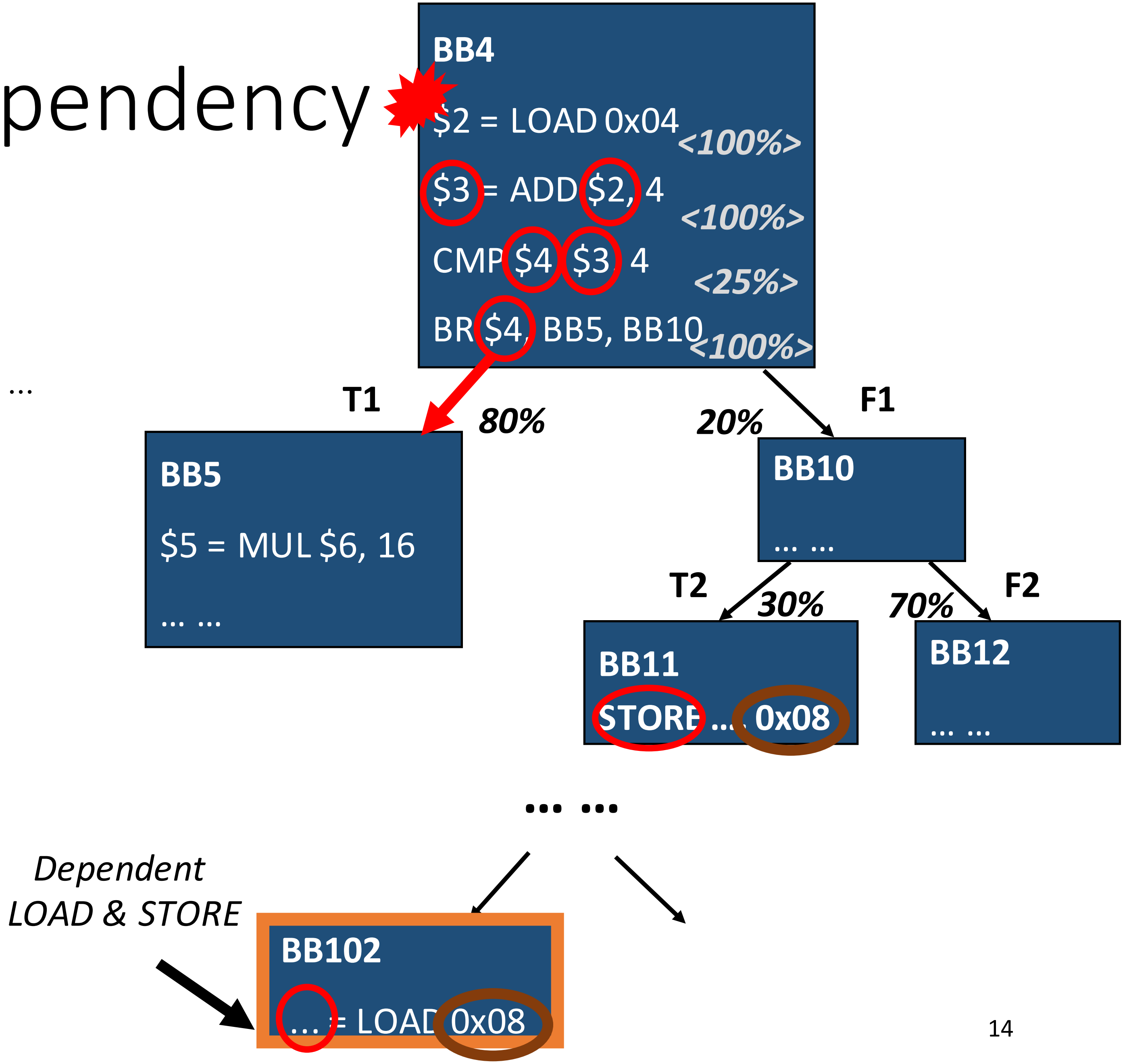


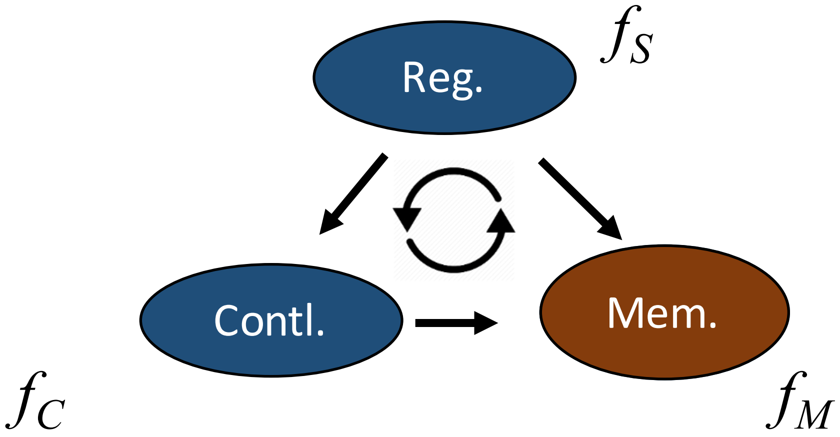


Trident: Memory-Dependency

$$P(I_n) = f_S(I_n) * f_C(I_{n2}) * f_S(I_{n3}) * f_C(I_{n4}) \;\dots\;\dots$$

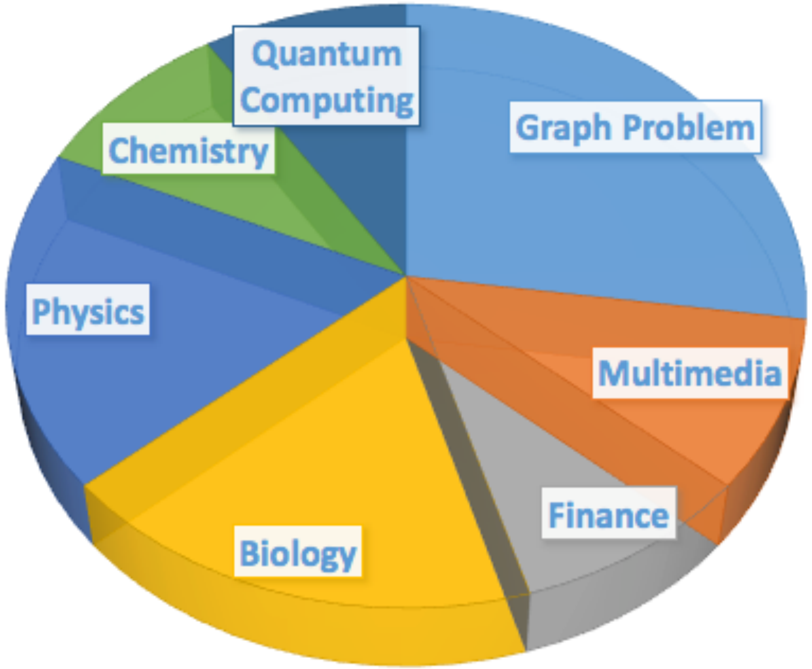
** n corresponds to the index of dynamic instructions*





Experimental Setup

- **Comparison with fault injection**
  - Accuracy
  - Speed (wall clock time)
- **Fault Model**
  - Single bit-flip injections – accurate [DSN’17]
  - Random insn. – one per program execution
- **Benchmarks**
  - 11 open-source benchmarks from various domains



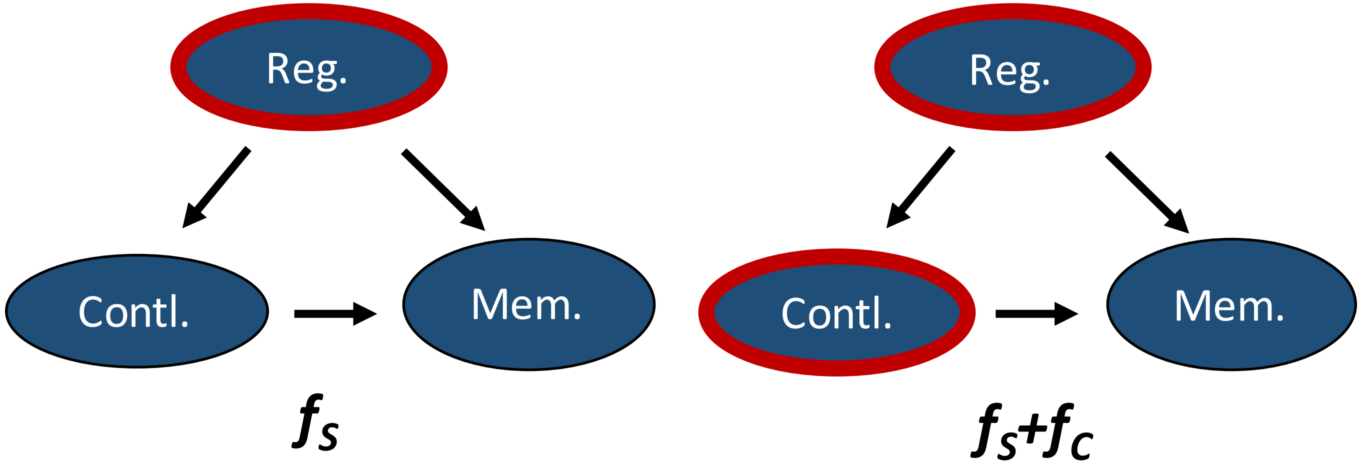
Benchmark Application Domains

Experimental Methodology

- **Created two simpler models**
  - Accuracy of each sub-model
  - As proxy to prior work
- **Baseline: Fault injection derived by LLFI [1]**
  - The closer SDC rate to fault injection, the better prediction

Reminder :

Goal is to predict SDC rate as per fault injection

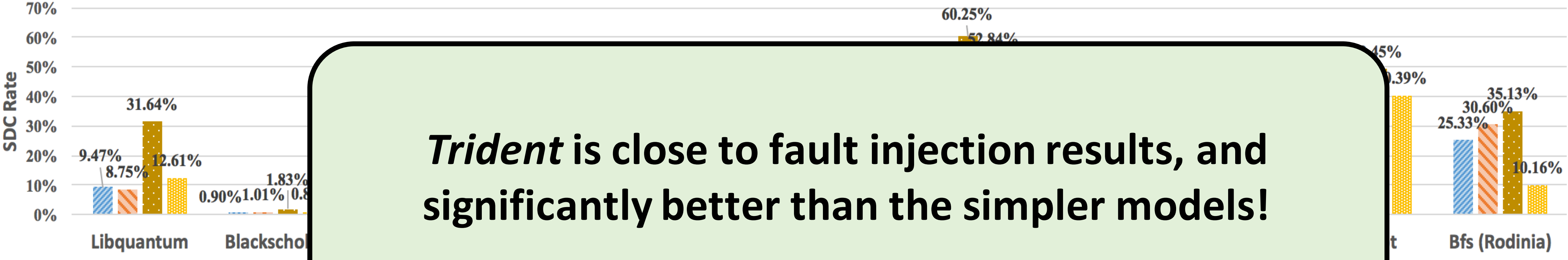


Two Simpler Models for Comparison

[1] LLVM Fault Injector [DSN'14]

Evaluation: Accuracy

Program SDC Rate; 3,000 Sampled Instructions; Error Bar: +/-0.07% ~ +/-1.76% at 95% Confidence Interval



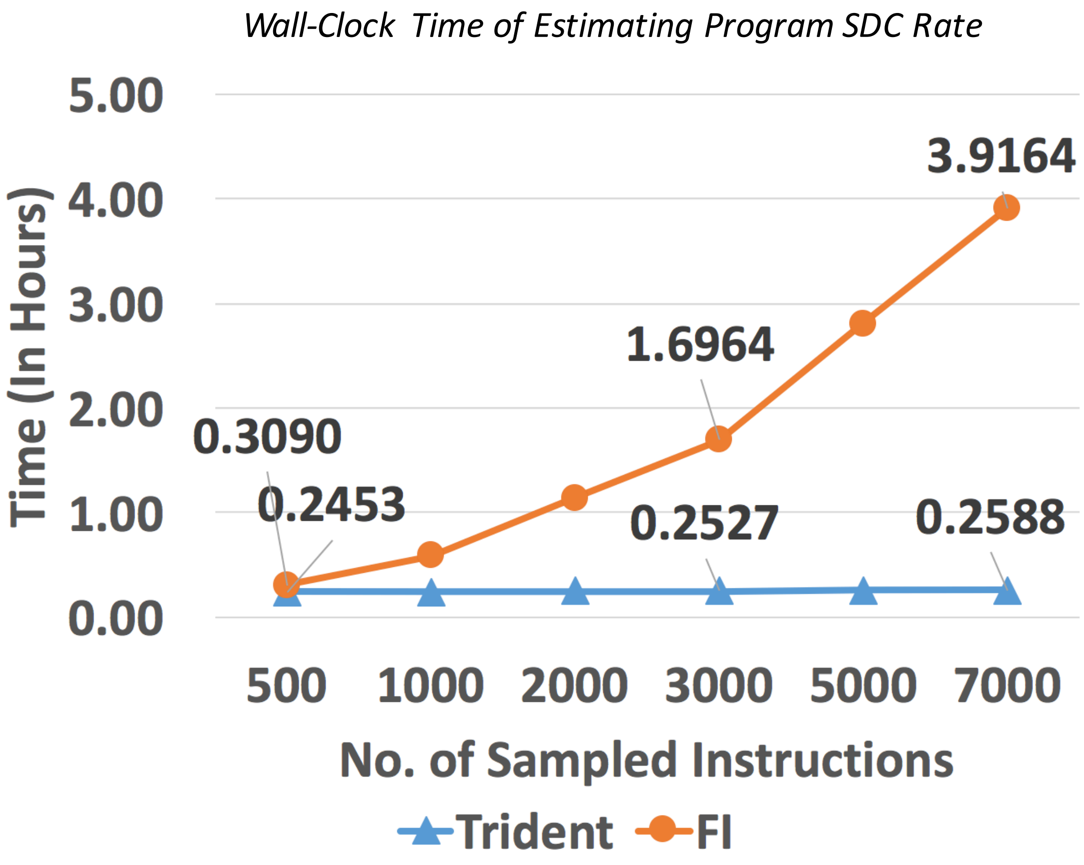
Trident **is close to fault injection results, and significantly better than the simpler models!**

- **Mean Absolute Error**
  - Trident: 4.75%
  - Simpler Models: 15.13% and 19.13%
- **t-Test on Individual Instructions**
  - Trident: 8 out of 11 are statistically indistinguishable
  - Simpler Models (f_S *and* f_S+f_C): Only 2 and 4

3,000 randomly sampled instructions for fault injection and the models

Evaluation: Speed

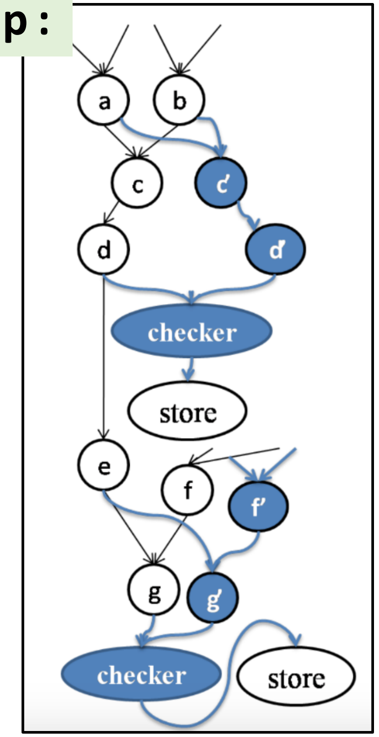
- **Program's Overall SDC Rate:**
  - 6.7x faster at 3,000 samples
- **Per-Instruction SDC Rate:**
  - On average, 380x faster at 100 samples per instruction
  - Benchmarks: FI takes nearly 100 hours whereas Trident takes <20 mins



***Trident* is faster than fault injection by 2 orders of magnitude!**

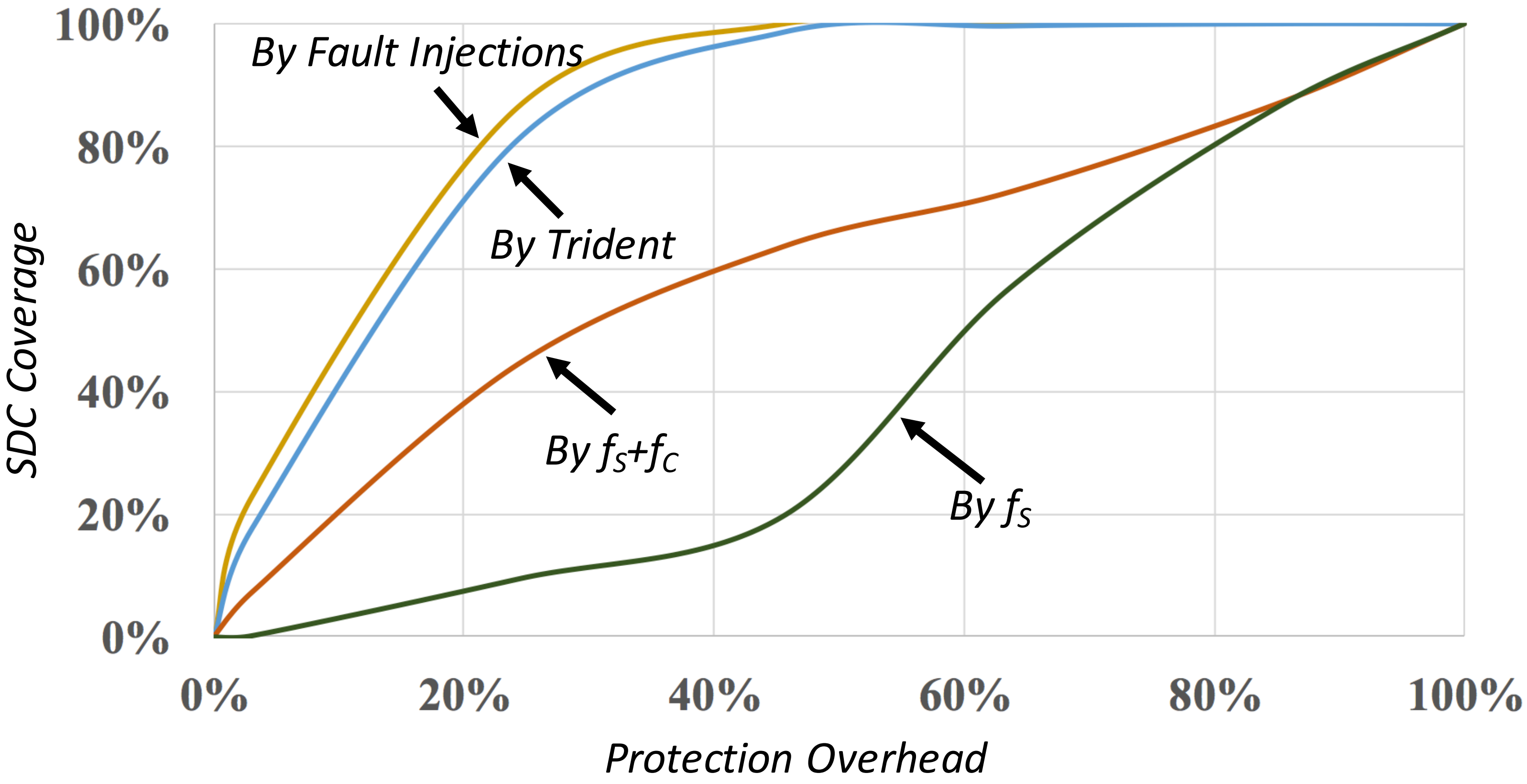
Use Case: Selective Instruction Duplication

Recap :



Selective Instruction Duplication

"The Golden Curve"



**Measured in Libquantum, SPEC*

Extension

- Understand how error propagation is affected by multiple inputs
- Extension for bounding SDC rate with multiple inputs

Session 6: Modeling and Verification
Wednesday, June 27th
“Modeling Input-Dependent Error Propagation in Programs”

Summary

- Fault injections are too slow to integrate into software development cycle
- *Trident* is both accurate and fast in predicting SDC rates
- Can guide selective protection of instructions in programs – comparable to fault injection in accuracy for fraction of cost
- Open Source: https://github.com/DependableSystemsLab/Trident

Guanpeng (Justin) Li
University of British Columbia (UBC)
gpli@ece.ubc.ca