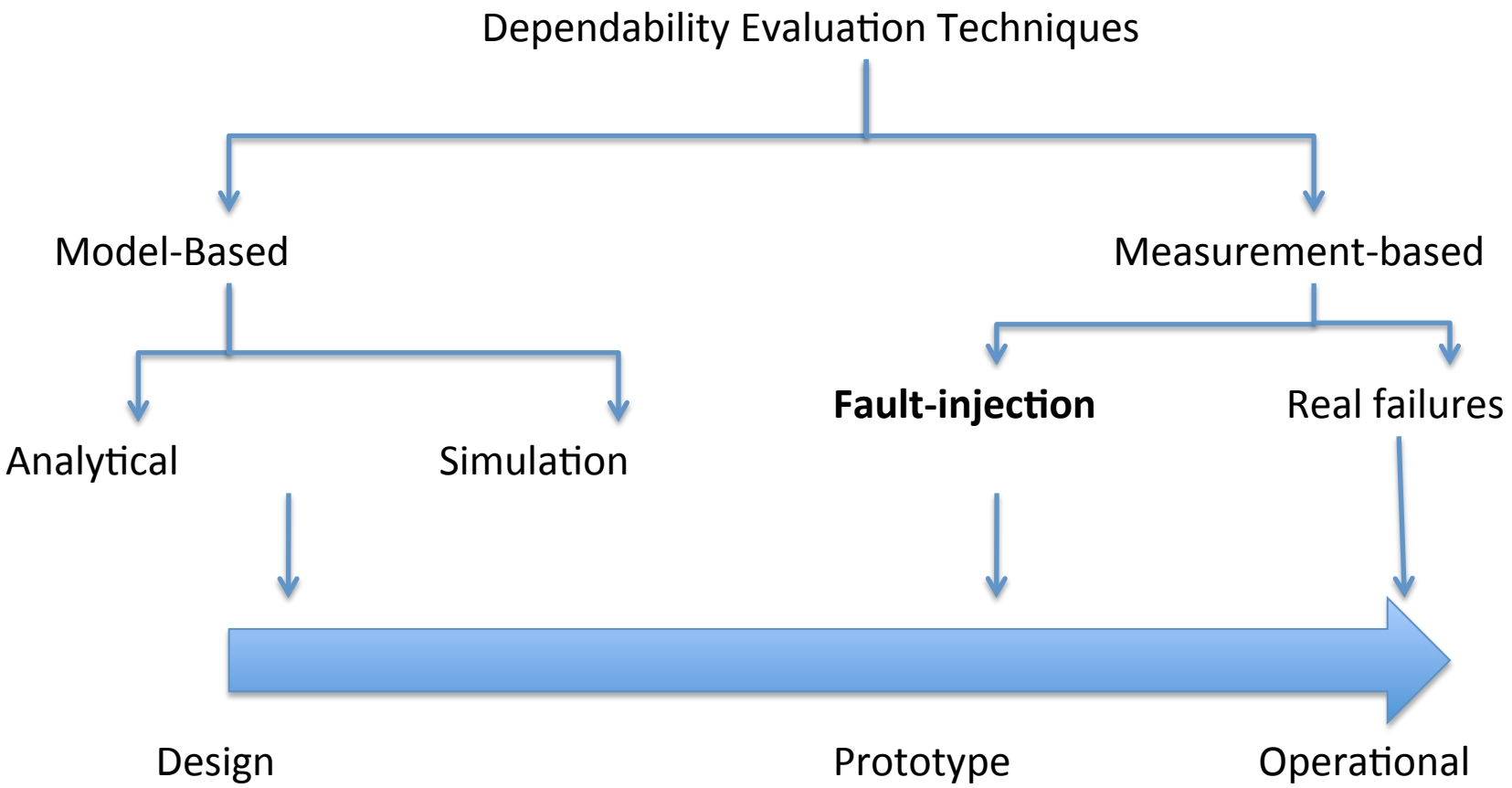
Fault Injection: Techniques, Tools and Trade-offs

EECE 513: Design of Fault-tolerant Systems

What will we learn ?

- Fault-injection: Motivation and means
- Fault-injection at different levels
- LLFI: Configurable Runtime Fault Injector

Dependability Evaluation



Fault-injection

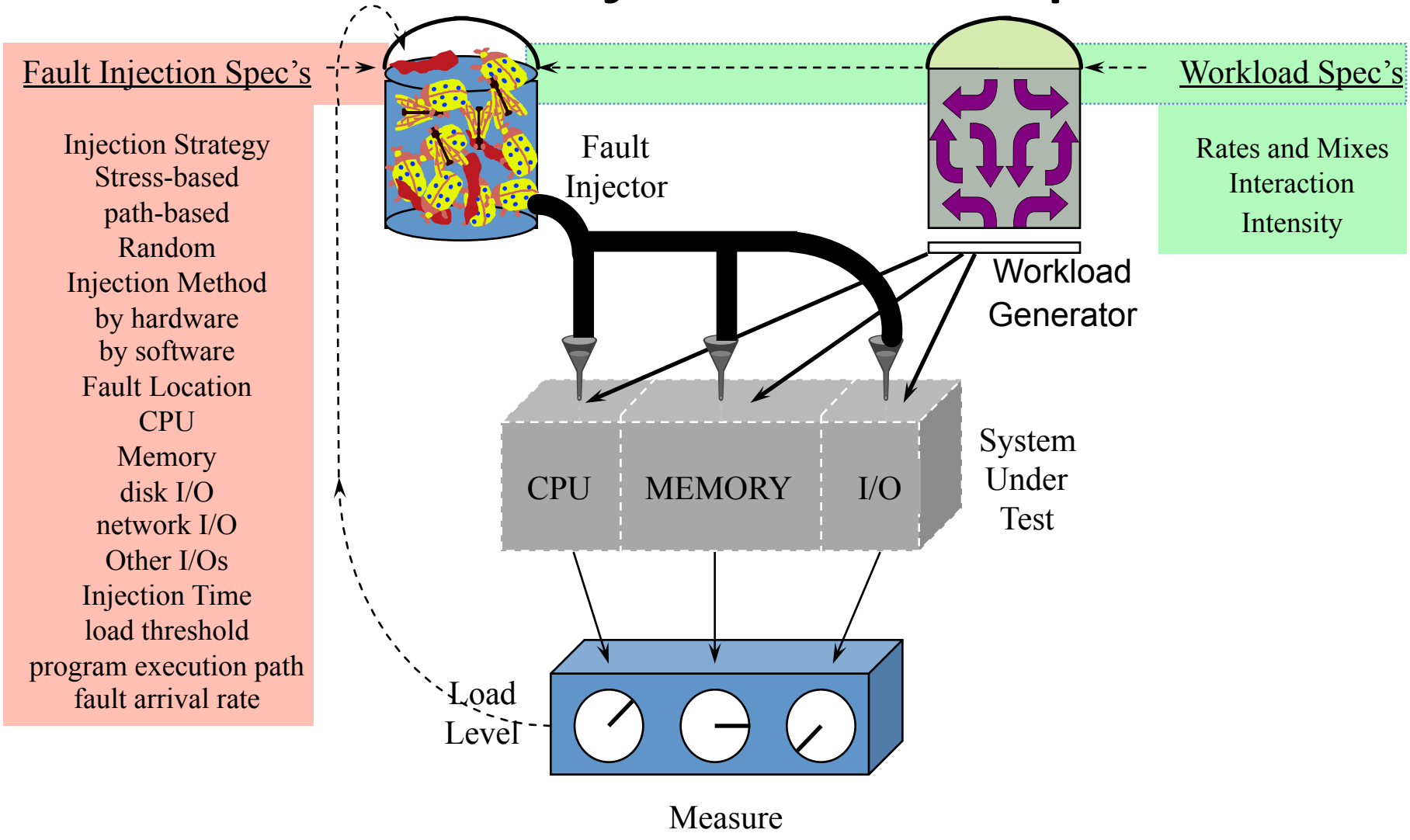
- Fault-injection (or fault-insertion) is the act of deliberately introducing faults into the system in a controlled and scientific manner, in order to study the system's response to the fault
  - Can be used to estimate coverage of dependability mechanisms (e.g., detection, recovery)
  - Also used to understand inherent fault tolerance
  - To obtain reliability estimates of the system prior to deployment (requires statistical projection)

Why fault-injection ?

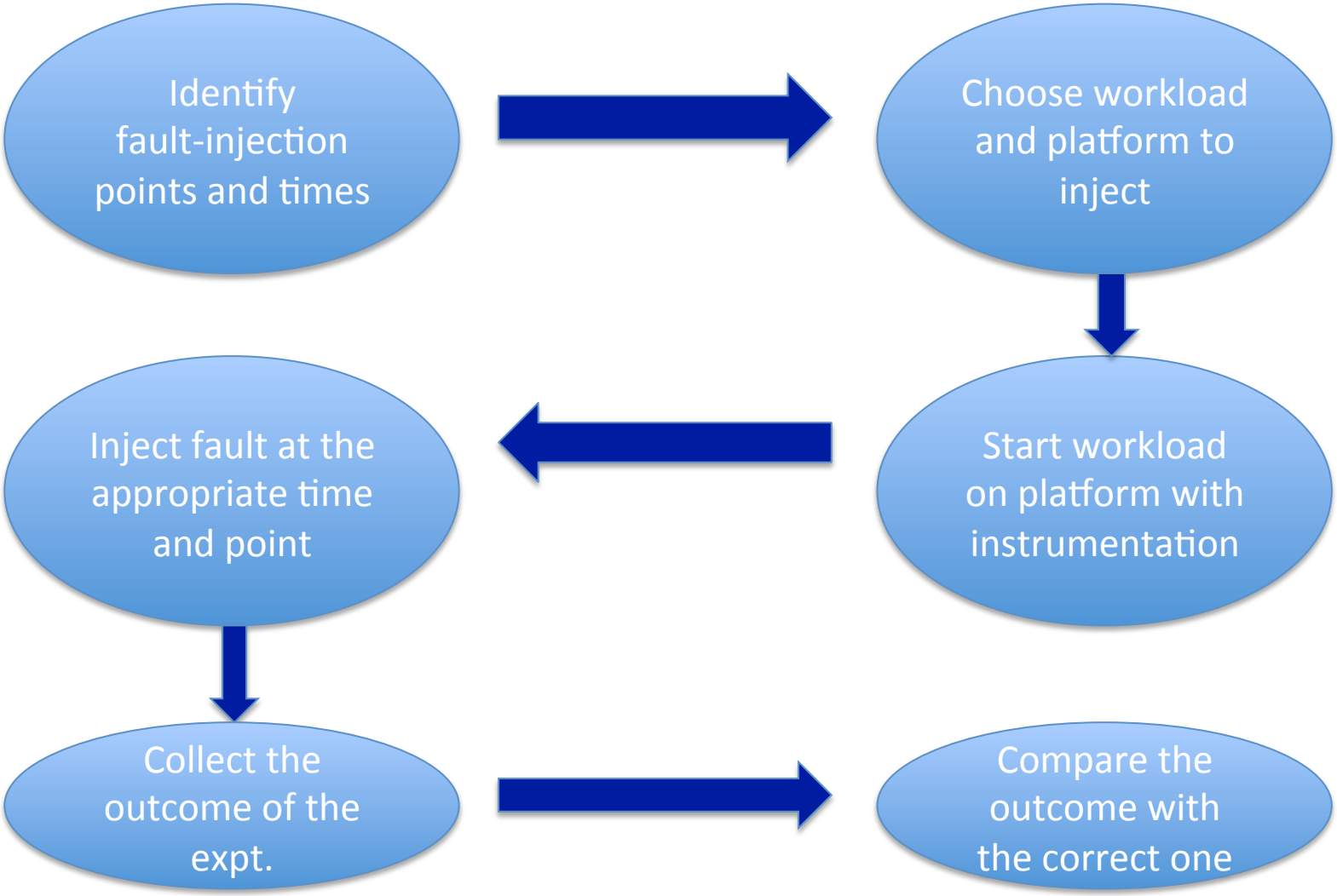
- **Versus Model-based**
  - More realistic, as it evaluates actual system
  - No need to worry about mathematical feasibility
  - No need to supply input parameters

- **Versus operational measurements**
  - Failures take a ***long*** time to occur and when they do, are often not reproducible or analyzable
  - Failures provide limited insight into what ***can*** go wrong
  - One has to wait until the system is deployed, which may be too late

Fault-Injection Setup



Fault-injection Steps

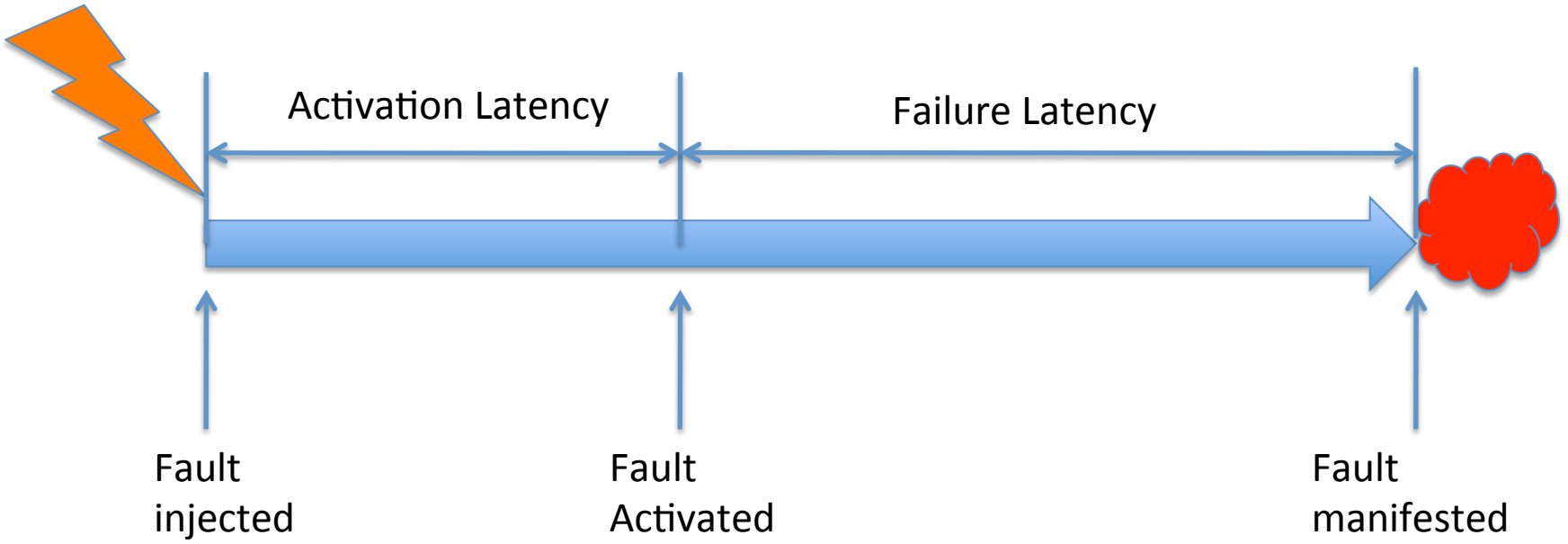


Fault-injection: Inputs/Outputs

- **Inputs**
  - Workload and platform to inject ?
  - When and where to inject ?
  - How many faults to inject (total) ?

- **Outputs**
  - How many faults were activated ?
  - How many faults cause a deviation of the outcome ?
  - What is the latency of manifestation ?

Measures to Compute



- What fraction of injected faults are activated ?
- What fraction of activated faults manifest as failures ?
- What are the average activation and failure latencies ?

Assumptions/Requirements

- A representative set of faults must be injected
  - Need to include enough faults to give confidence in the measures being studies
- Only one or controlled no. of faults injected
  - Ability to map the outcome to a set of faults
- Need to have a specification of correct behavior to distinguish incorrect outcomes
  - May need to determine golden run ahead of time

What will we learn ?

- Fault-injection: Motivation and means
- Fault-injection at different levels
- LLFI: Configurable Runtime Fault Injector

Levels of Fault-Injection

- Fault-injection can be performed at multiple levels, from hardware to software

- **Three things to consider in choosing level**
  - Type of fault to inject (e.g., stuck at faults easier to inject in the hardware than in software)
  - Speed of injection (e.g., h/w simulation slower than real execution, though direct h/w probes possible)
  - Intrusiveness (e.g., probing hardware result in physical modifications that change the system's characteristics)

Fault-Injection and Fault-Models

Hardware

- Open
- Bridging
- Stuck-at
- Power Surge
- Spurious Current
- Bit-flip

Software

- Storage Data Corruption
  - Registers, Memory, Disk
- Communication data corruption
  - CRC errors, Bus Errors
- Software defect emulation
  - Machine code corruption, source code mutation

Hardware fault-injection

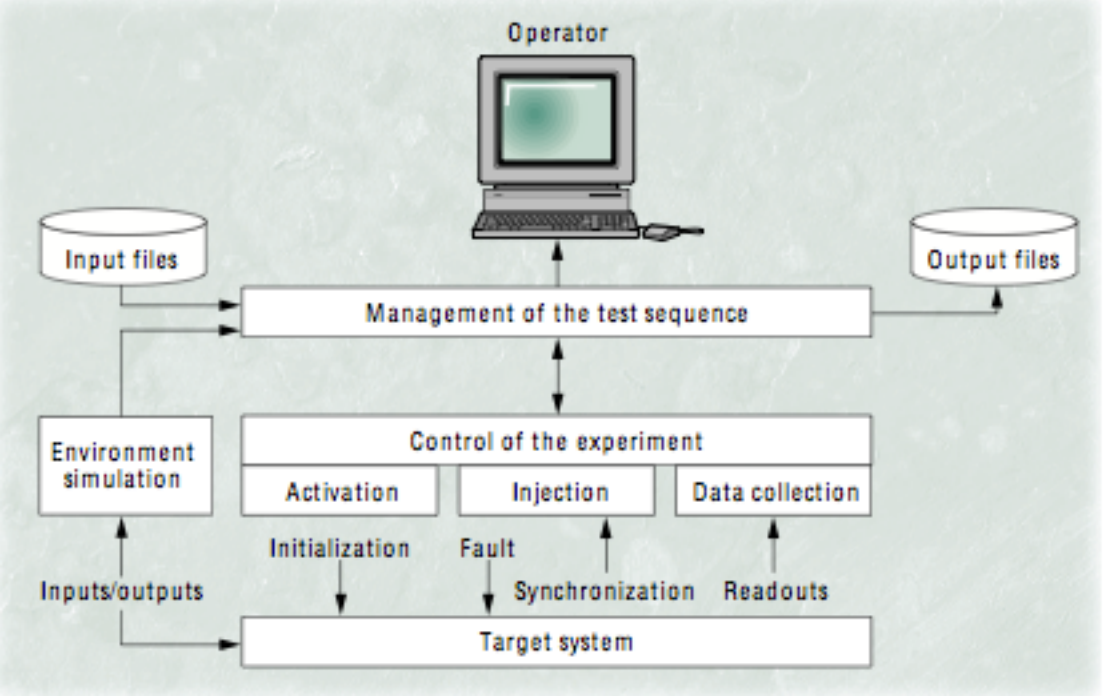
Contact-based

- **Active Probes:** Alters the current via probes attached to the pins
  - Usually limited to stuck-at-faults, though bridging faults can also be modeled
  - Care must be taken to not damage the pins
- **Socket based:** Insert a socket between the target hardware and the circuit board
  - Can inject stuck-at or other logical faults

Non-contact based

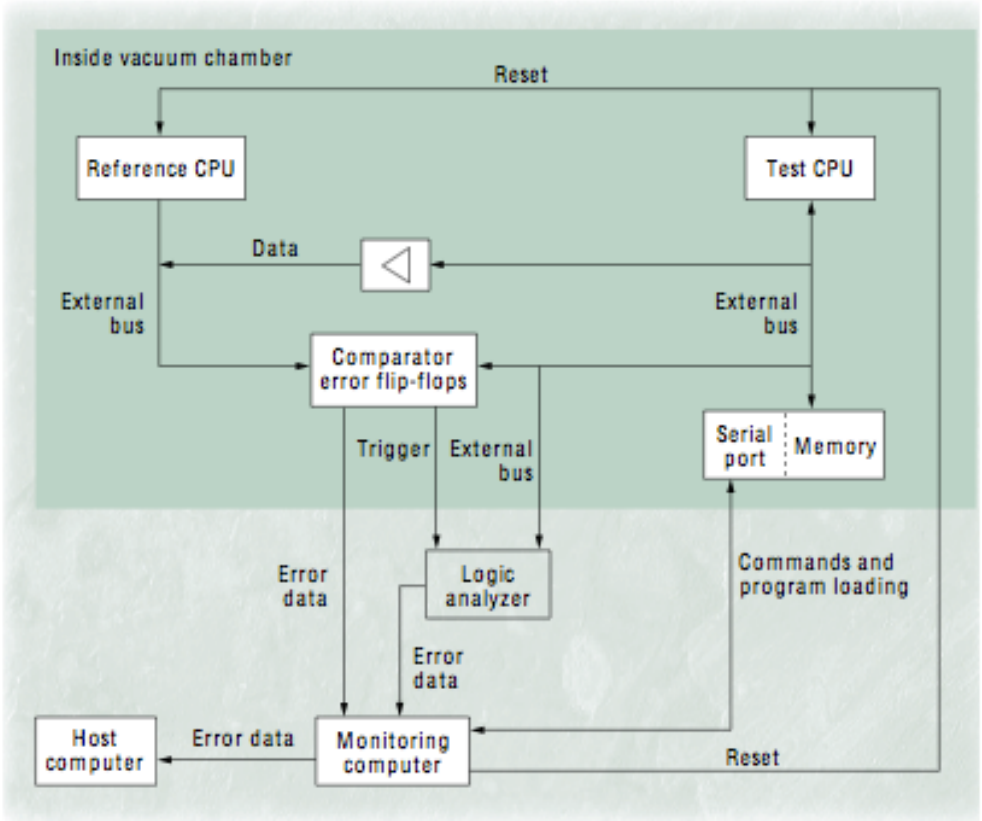
- Heavy-ion Radiation: Put the chip in an accelerator beam (e.g., TRIUMF)
  - Difficult to control and reproduce
  - But injects realistic faults
  - No restriction on where faults can be injected
- **Placing chip in an EM field**
  - Can lead to permanent damage

H/W Fault-Injection: Example (Contact Based)



Messaline from CNRS [Arlat'1990]: Can perform probe-based and socket-based injection. used for evaluating safety-critical systems such as railway control system

Hardware Fault-Injection (Non-contact Based)



FIST from Chalmers [Karlson'1995]: Used a Vaccum chamber in which an ionizing source was placed. A second non-faulty processor was used for state comparision.

Software-based Fault-Injection (SWIFI)

Pros

- Do not require expensive hardware modifications
- Can target applications and OS errors
- Many hardware faults do not require probes, e.g, register data corruption

Cons

- Restricted to inject only faults that S/W can see
- May perturb the workload that is running on the system, resulting in missing many heisenbugs
- Coarser-grained time resolution than h/w

SWIFI: Types

Compile-time

- Modify source code or machine code of the program prior to execution
- Can be used to model software defects
- Requires going thro' compile-run cycle each time

Runtime

- Modify the program or its data during runtime
- Can be done through the debugger, kernel or with support from compiler
- No need to go through compile-run cycle each time

Compile-time Injection

- Modify program's code prior to execution
  - Model hardware transient faults in machine code
  - Also, allows for modeling of software errors
  - Typically only inject into the first dynamic instance of an instruction
- Main advantage: Take advantage of static analysis of the code to customize the injection

Runtime Injection

- Advantages
  - Can inject faults without recompiling - speed
  - Faults can occur deeper in the execution. e.g., one-millionth iteration of a loop
  - Fault can depend on runtime conditions. e.g., if memory usage exceeds a threshold, inject fault

What will we learn ?

- Fault-injection: Motivation and means
- Fault-injection at different levels
- LLFI: Configurable Runtime Fault Injector

Why yet another fault injector?

- **Difficult to customize existing injectors**
  - Inject into specific instructions
  - Inject into a specific variable
  - Inject into specific code constructs

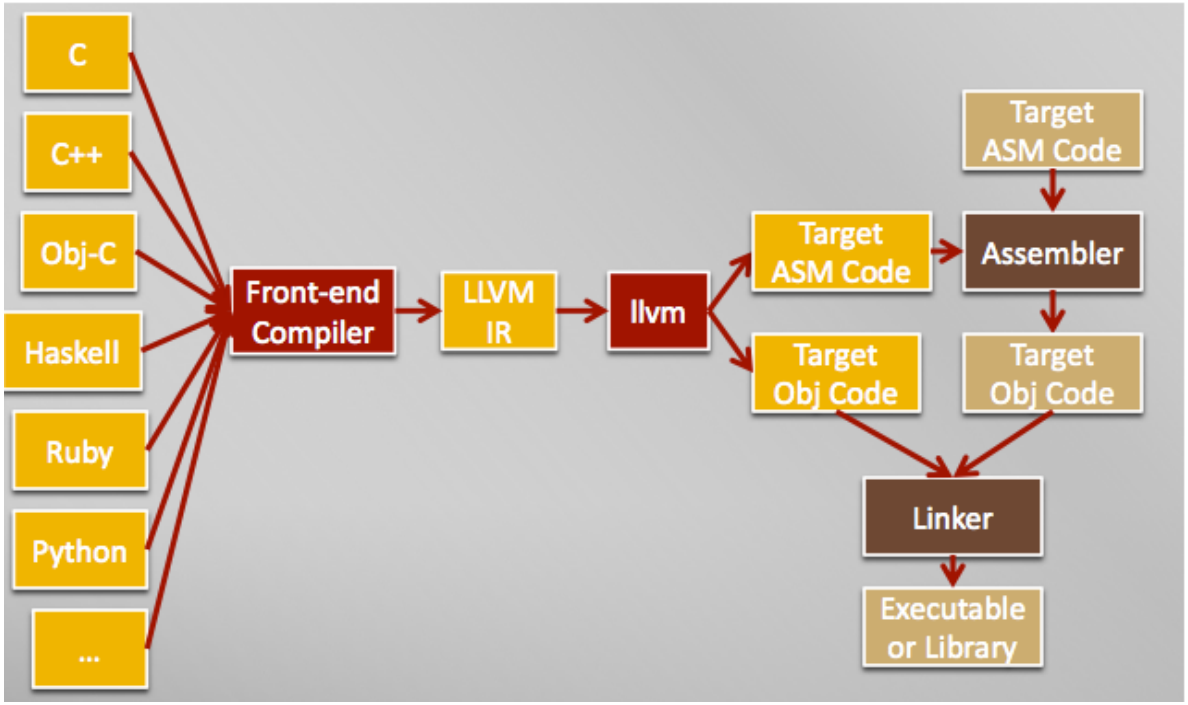
- **Difficult to understand the results**
  - Difficulty in fault injection customization
  - Difficult to study the propagation of errors
  - Difficult to map result back to source code

LLFI

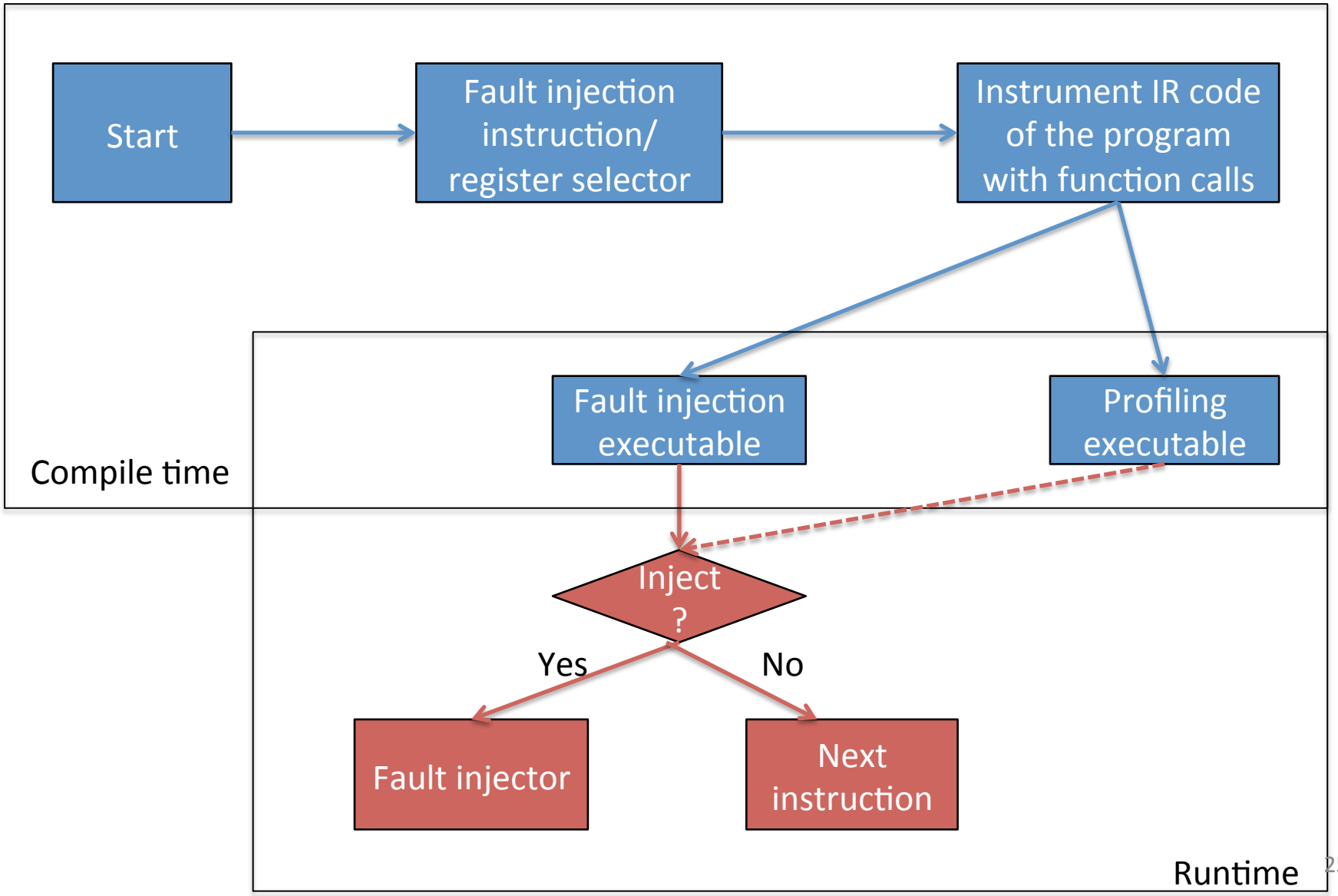
- **A fault injector based on LLVM**
  - Intermediate representation (IR) level injection
- **Features**
  - Easy to customize the fault injection
  - Easy to analyze fault propagation
  - Accurate compared to assembly level injection

Why LLVM Compiler ?

- Supports wide variety of front- and back-ends
- Provides high-level features in the IR code



How does LLFI work?



Factorial Example: Original

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
  %"alloca point" = bitcast i32 0 to i32
  %0 = getelementptr inbounds i8** %argv, i64 1
  %1 = load i8** %0, align 1
  %2 = call i32 (...)* @atoi(i8* %1) nounwind
  br label %bb1

bb:                                               ; preds = %bb1
  %3 = mul nsw i32 %fact.0, %i.0
  %4 = add nsw i32 %i.0, 1
  br label %bb1

bb1:                                              ; preds = %bb, %entry
  %i.0 = phi i32 [ 1, %entry ], [ %4, %bb ]
  %fact.0 = phi i32 [ 1, %entry ], [ %3, %bb ]
  %5 = icmp sle i32 %i.0, %2
  br i1 %5, label %bb, label %bb2

bb2:                                              ; preds = %bb1
  %6 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr
       inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 %fact.0) nounwind
  br label %return

return:                                           ; preds = %bb2
  ret i32 undef
}
```

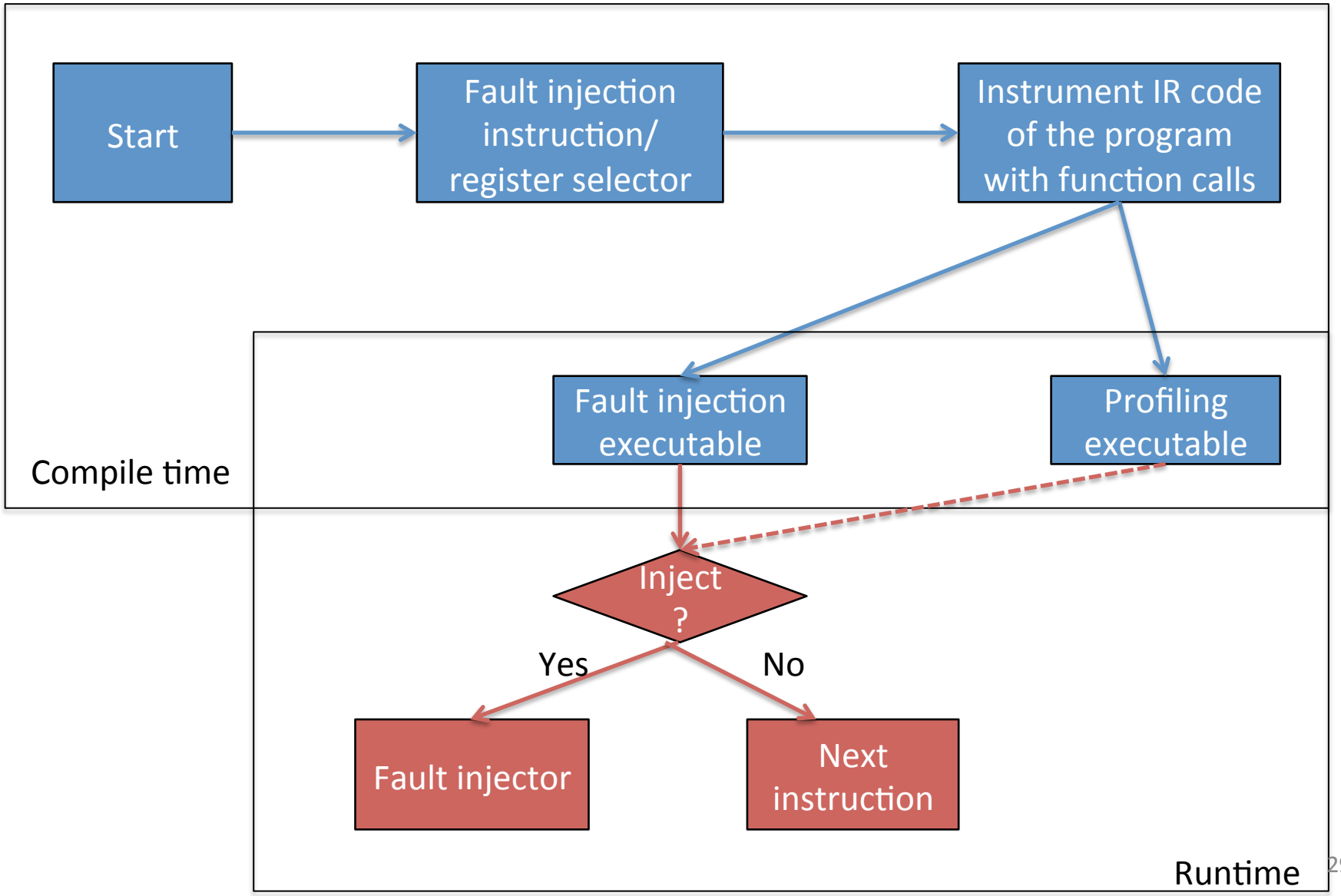
Factorial Example: Instrumented

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
  call void @initInjections(i8* getelementptr
        inbounds ([21 x i8]* @NameStr, i32 0, i32 0))
  %"alloca point" = bitcast i32 0 to i32
  %fi2 = call i32 @injectFault0(i32 2, i32 0, i32 %"alloca point")
  %0 = getelementptr inbounds i8** %argv, i64 1
  %fi3 = call i8** @injectFault1(i32 3, i32 0, i8** %0)
  %1 = load i8** %fi3, align 1
  %fi4 = call i8* @injectFault2(i32 4, i32 0, i8* %1)
  %2 = call i32 (...)* @atoi(i8* %fi4) nounwind
  %fi5 = call i32 @injectFault0(i32 5, i32 0, i32 %2)
  br label %bb1

bb:                                               ; preds = %bb1
  %3 = mul nsw i32 %fi8, %fi1
  %fi6 = call i32 @injectFault0(i32 6, i32 0, i32 %3)
  %4 = add nsw i32 %fi1, 1
  %fi7 = call i32 @injectFault0(i32 7, i32 0, i32 %4)
  br label %bb1

bb1:                                              ; preds = %bb, %entry
  %i.0 = phi i32 [ 1, %entry ], [ %fi7, %bb ]
  %fact.0 = phi i32 [ 1, %entry ], [ %fi6, %bb ]
  %fi8 = call i32 @injectFault0(i32 8, i32 0, i32 %fact.0)
  %fi1 = call i32 @injectFault0(i32 1, i32 0, i32 %i.0)
  %5 = icmp sle i32 %fi1, %fi5
  %fi9 = call i1 @injectFault3(i32 9, i32 0, i1 %5)
  br i1 %fi9, label %bb, label %bb2

bb2:                                              ; preds = %bb1
  %6 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr
        inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 %fi8) nounwind
  %fi10 = call i32 @injectFault0(i32 10, i32 0, i32 %6)
  br label %return

return:                                           ; preds = %bb2
```

Features of LLFI

- **Easy to customize the fault injection**
- Easy to analyze the fault propagation
- Accurate compared to assembly level injection

Easy Fault Injection Customization



Easy Fault Injection Customization

- Fault injection instruction selector
  - Based on instruction type
    - Include: add + cmp
    - Include: all; Exclude: load
  - Based on custom instruction selector
  - Include backward/forward trace

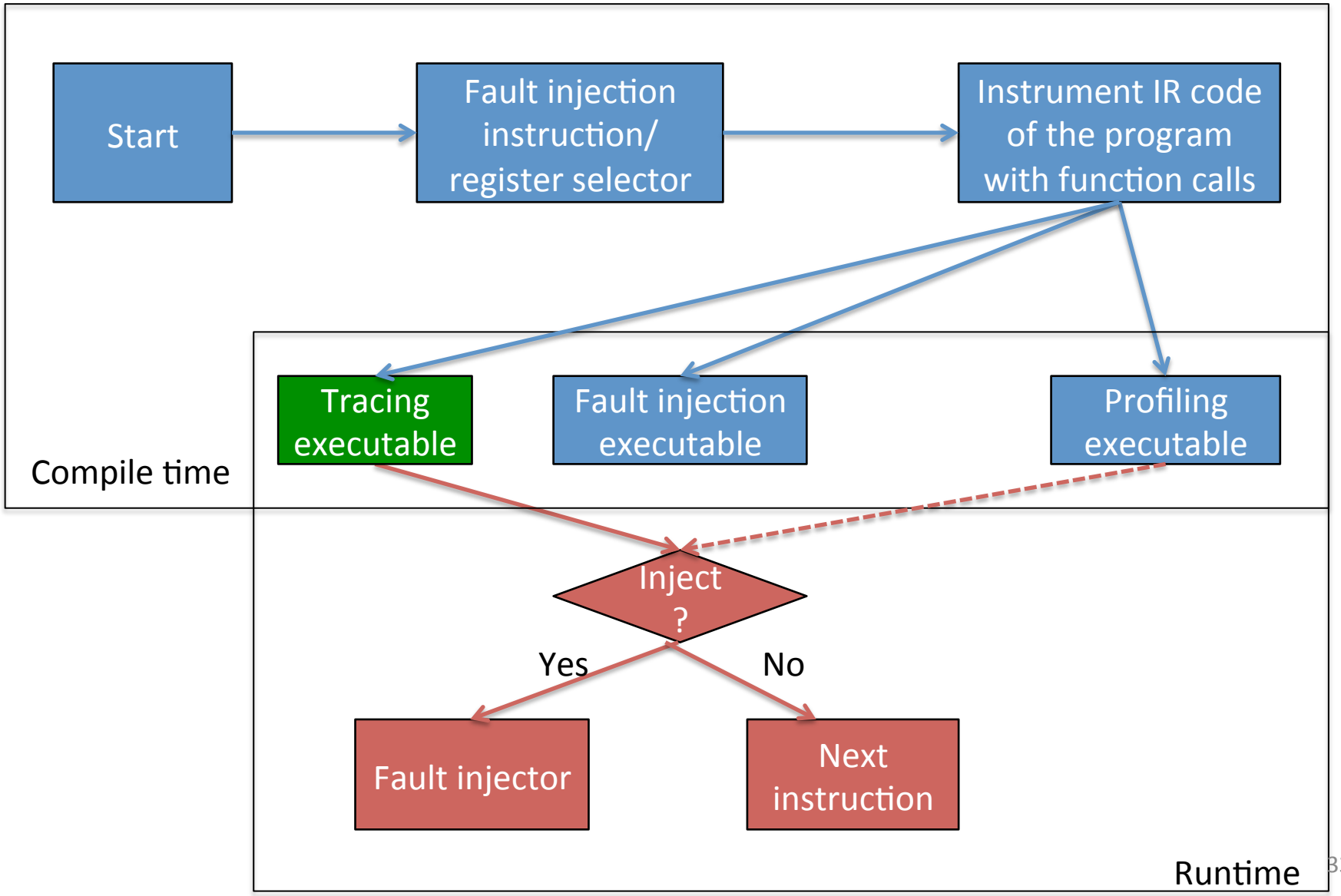
Easy Fault Injection Customization

- Fault injector
  - Common fault injectors
    - Bit-flip, stuck-at-0/1, etc.
  - Custom fault injectors
    - Specified by user as C function

Features of LLFI

- Easy to customize the fault injection
- **Easy to analyze the fault propagation**
- Accurate compared to assembly level injection

Easy Analysis

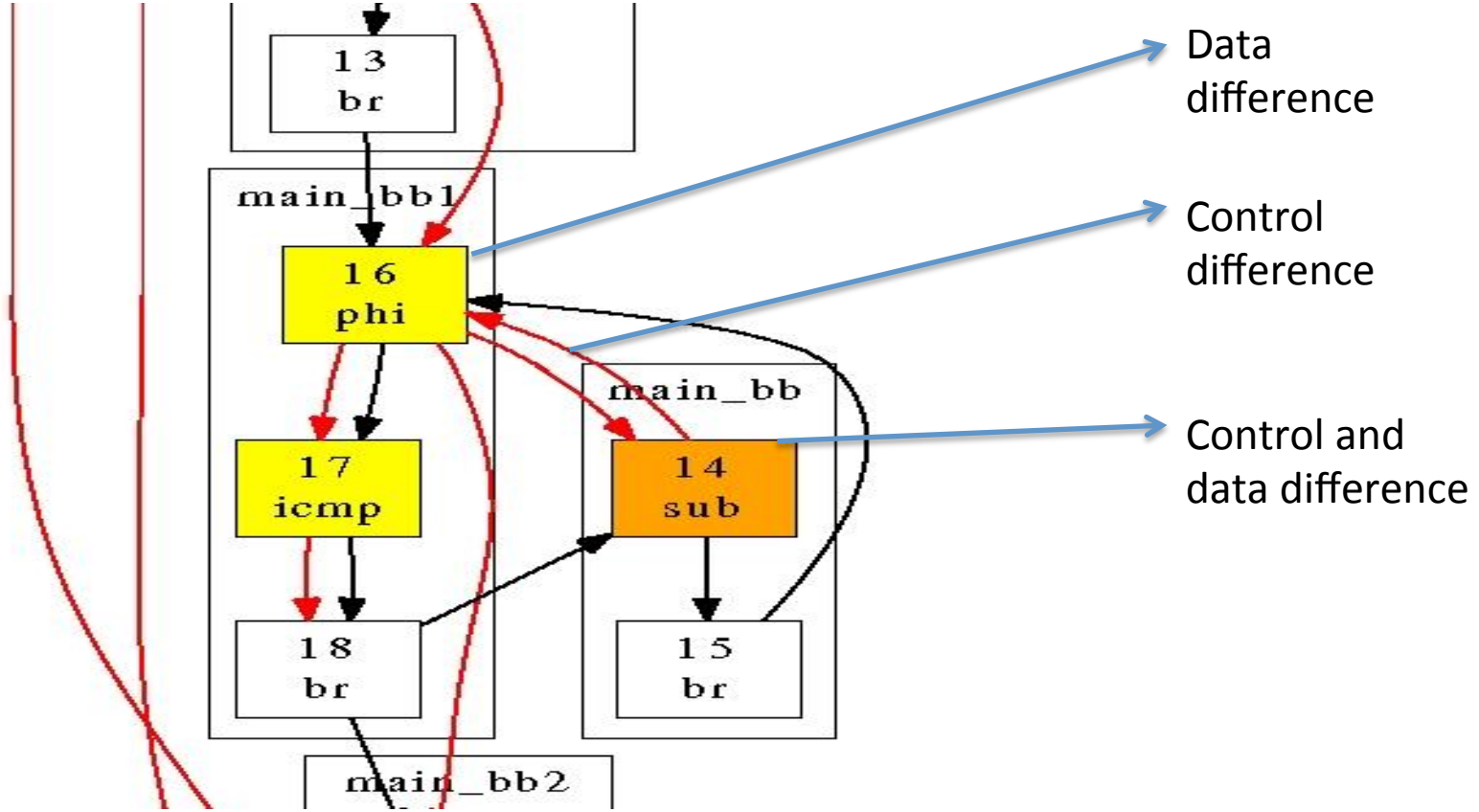


Easy Analysis

- Trace the value of every instruction
  - Obtain golden run and fault injection run
  - Can include forward and backward dependencies
  - Can limit the trace for performance reasons
- Perform a comparison
  - Data diff:
    - Instruction ID: 20/add: val 3 => 11
  - Control diff:
    - Instruction ID: 22/cmp: 22 -> 23 => 22 -> 24

Easy Analysis

- Graphical output of trace differences as dot file



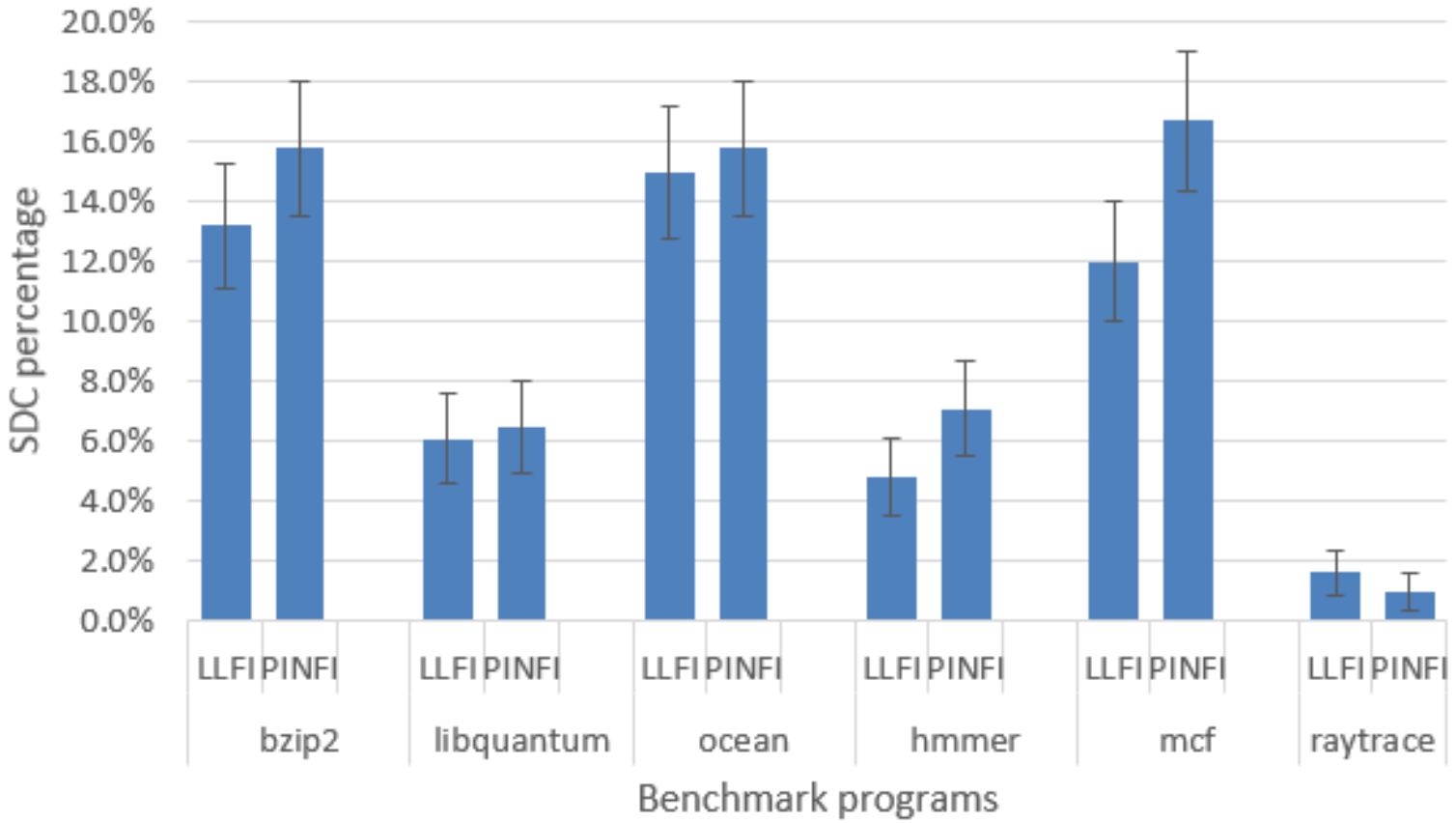
Features of LLFI

- Easy to customize the fault injection
- Easy to analyze the fault propagation
- **Accurate compared to assembly level injection**

Experimental Setup

- Compared LLFI with assembly language level fault injection implemented using PIN tool
- Used six benchmarks from SPEC, PARSEC to perform fault-injection experiments on both
- Classified results in crashes, SDCs, and benign

Accuracy Results: SDCs



Difference in SDC rate between LLFI and PIN < 5%

Summary and Ongoing Work

- LLFI[1]
  - Easy to customize your fault injection
  - Easy to analyze the result
  - Accurate compared to assembly code injection

- Ongoing Work
  - GUI to choose fault configuration options (in beta)
  - Extension to inject into multi-threaded programs

1. https://github.com/DependableSystemsLab/LLFI

What will we learn ?

- Fault-injection: Motivation and means
- Fault-injection at different levels
- LLFI: Configurable Runtime Fault Injector