FITL: Extending LLVM for the Translation of Fault-Injection Directives

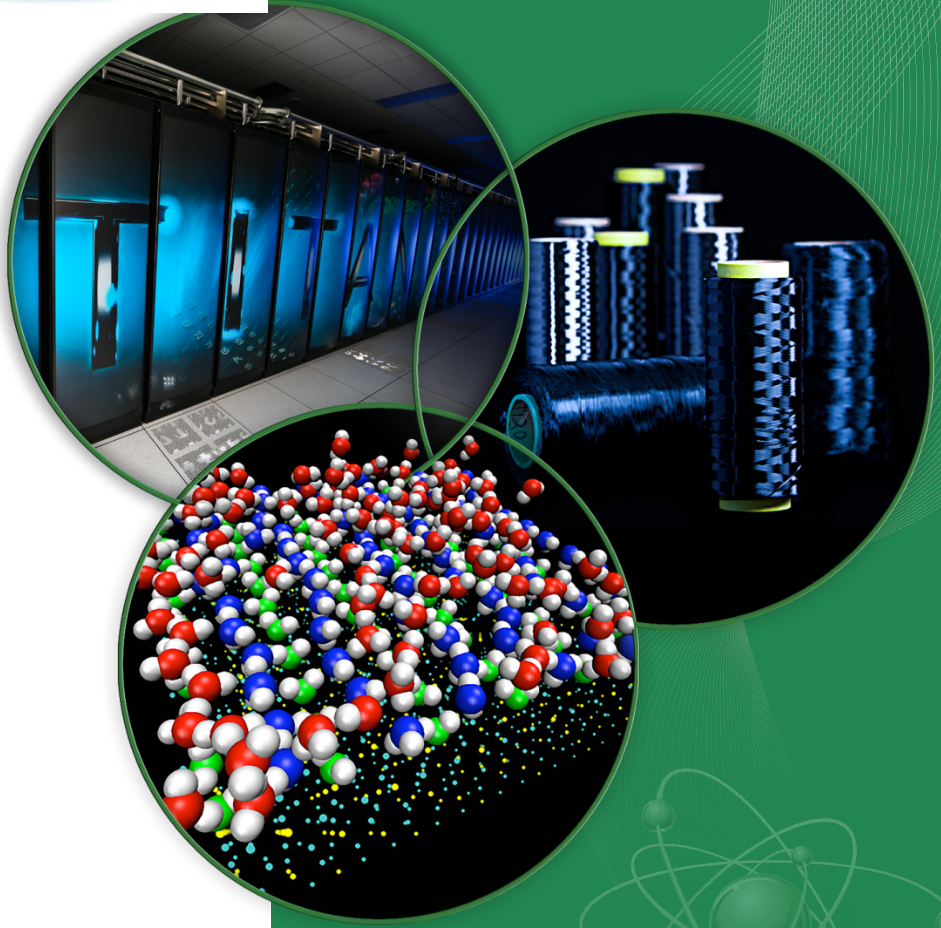
Joel E. Denny

Seyong Lee

Jeffrey S. Vetter

November 15, 2015

LLVM HPC @ SC15



JICS
Joint Institute for
Computational Sciences

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

http://ft.ornl.gov dennyje@ornl.gov

OAK RIDGE
National Laboratory

Outline

- Background & Motivation
- Design & Contributions
- Fault-Injection Pragmas for C
- FITL Extensions to LLVM
- Case Studies

Background & Motivation

Background: Fault Injection

- Emulate hardware faults to study application resiliency
- Example causes of faults:
  - Cosmic rays, alpha particles
  - Thermal effects
  - Noise
- Types of faults:
  - Permanent faults: we do not model this
  - Transient faults: modeled as single bit flips
- Impact of transient faults:
  - Benign fault – no impact on application
  - Crash – obvious failure of application
  - Silent data corruption (SDC)

Motivation: HPC and Resilience

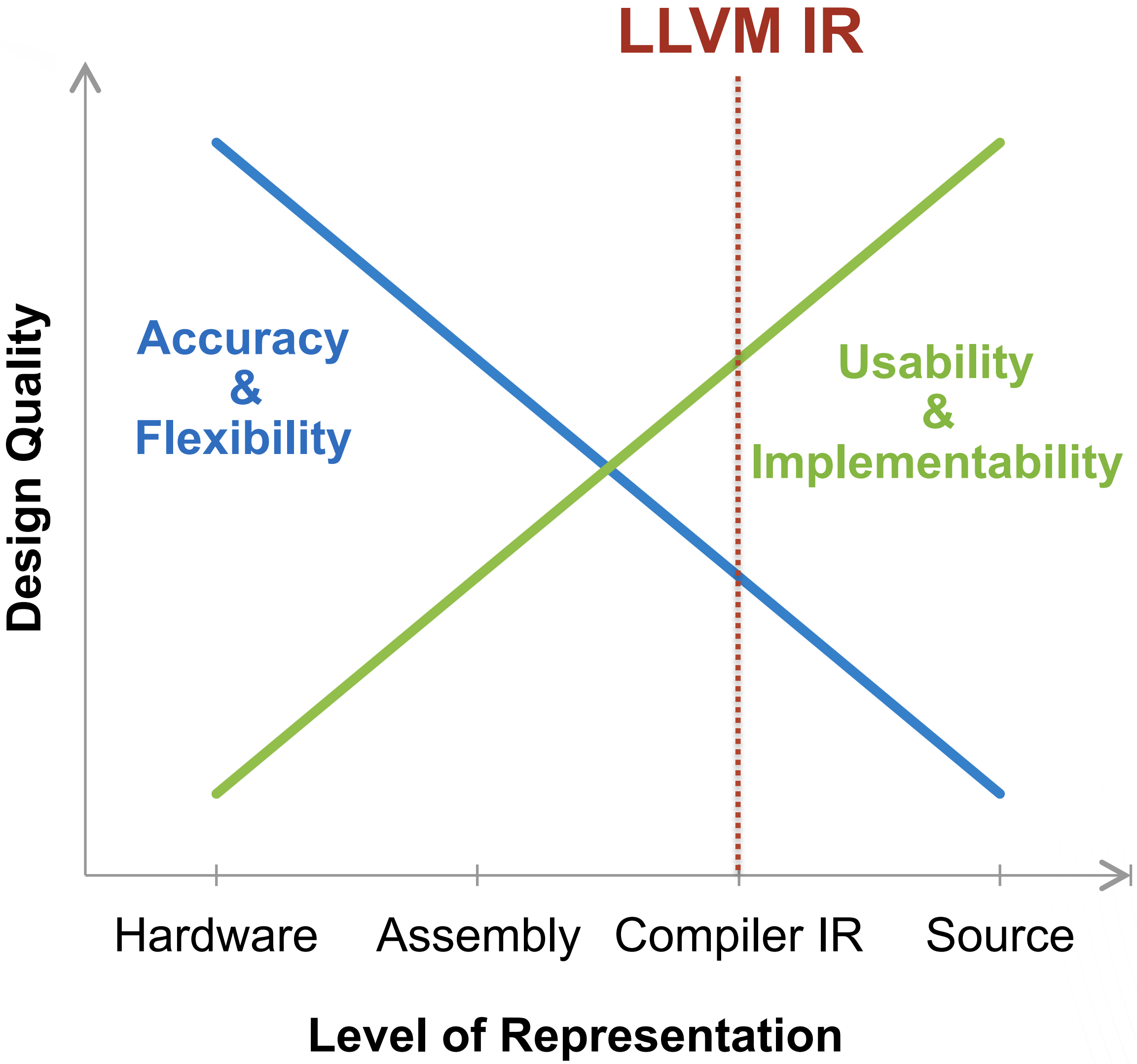
- HPC systems evolving toward exascale:
  - Millions of computational units, memory units, sockets
  - New power reduction strategies needed
  - Frequency of hardware faults will grow dramatically
- Prediction:
  - Mean time to failure will be too short for existing resiliency solutions
  - Checkpoint, restart, hardware-only solutions will not be sufficient
  - Resilience must be addressed in software stack
  - Hardware faults exposed to application level
- Need a way to study application resilience

Motivation: Fault Injector Tool Design

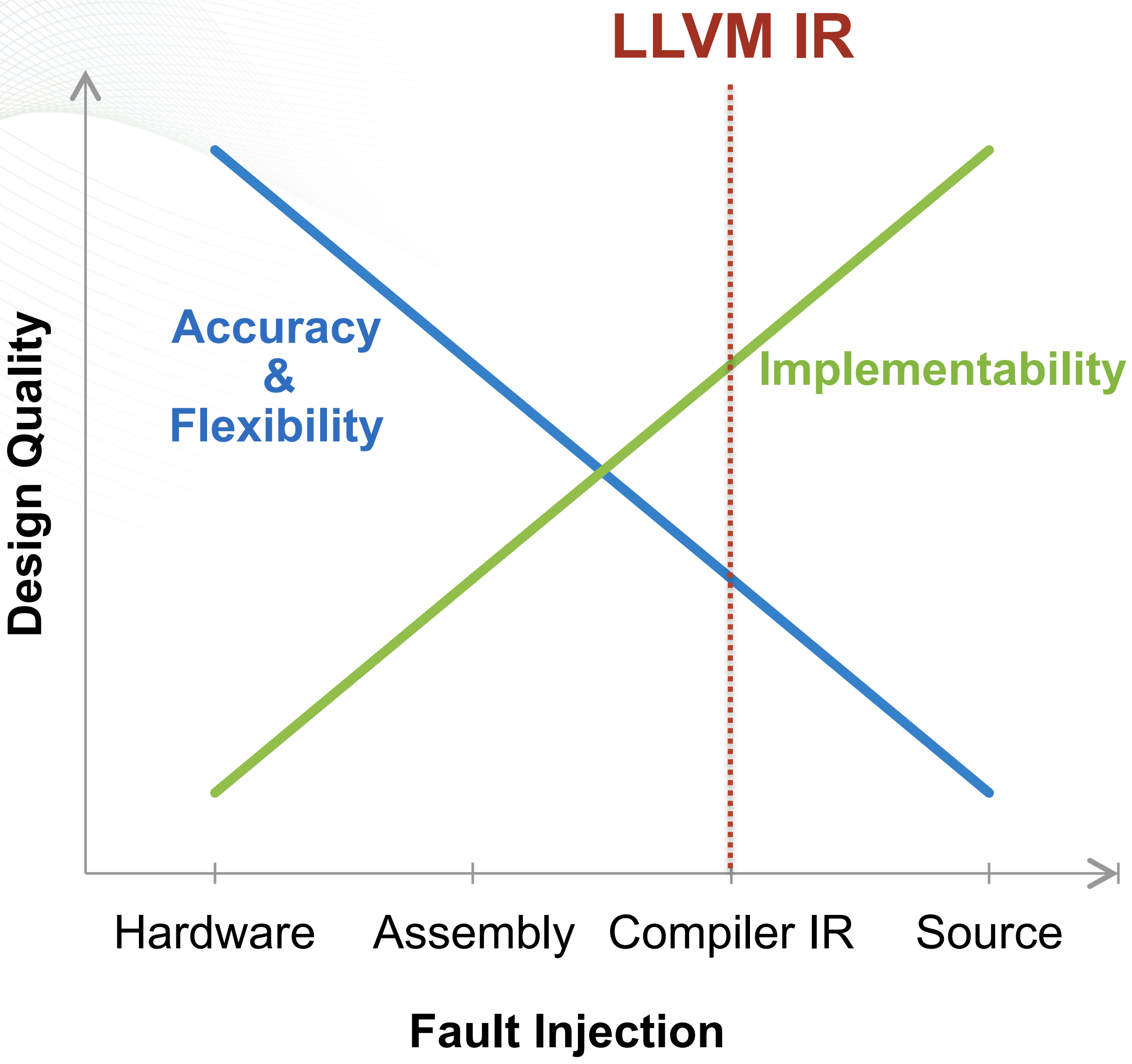
- Fault injector tool use cases:
  - Study resiliency of applications to develop new resilience mechanisms
  - Unit testing and debugging framework for resiliency
- Design qualities:
  - Flexibility: what kinds of hardware fault scenarios can we emulate?
  - Accuracy: are the simulations realistic?
  - Usability: how easy is it to configure studies and understand results?
  - Implementability: how easy is it to implement?
- Design variables:
  - Level of representation at which fault injector operates
  - Level of representation at which user configures fault injection

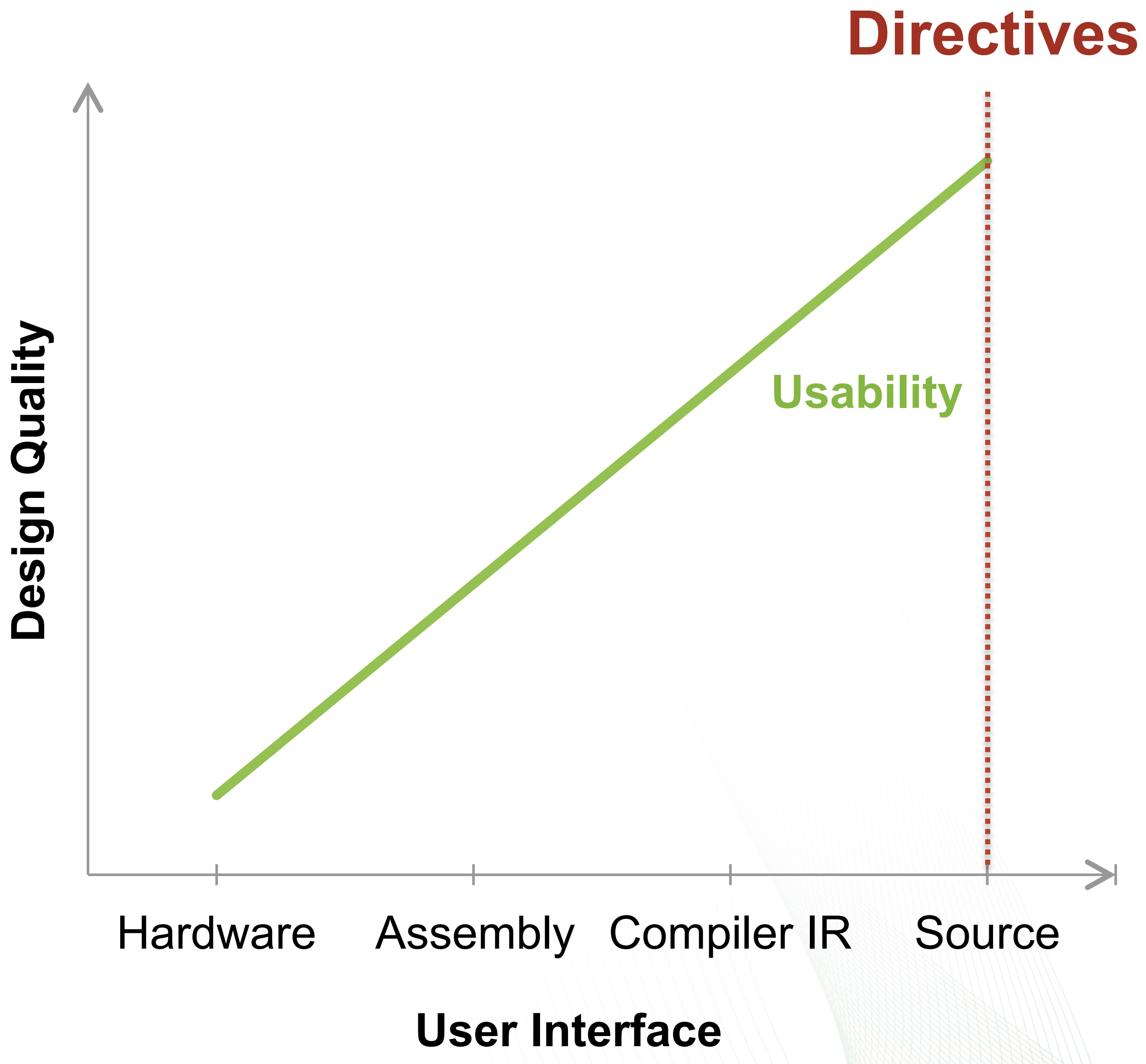
Design & Contributions

Design Quality vs. Level of Representation

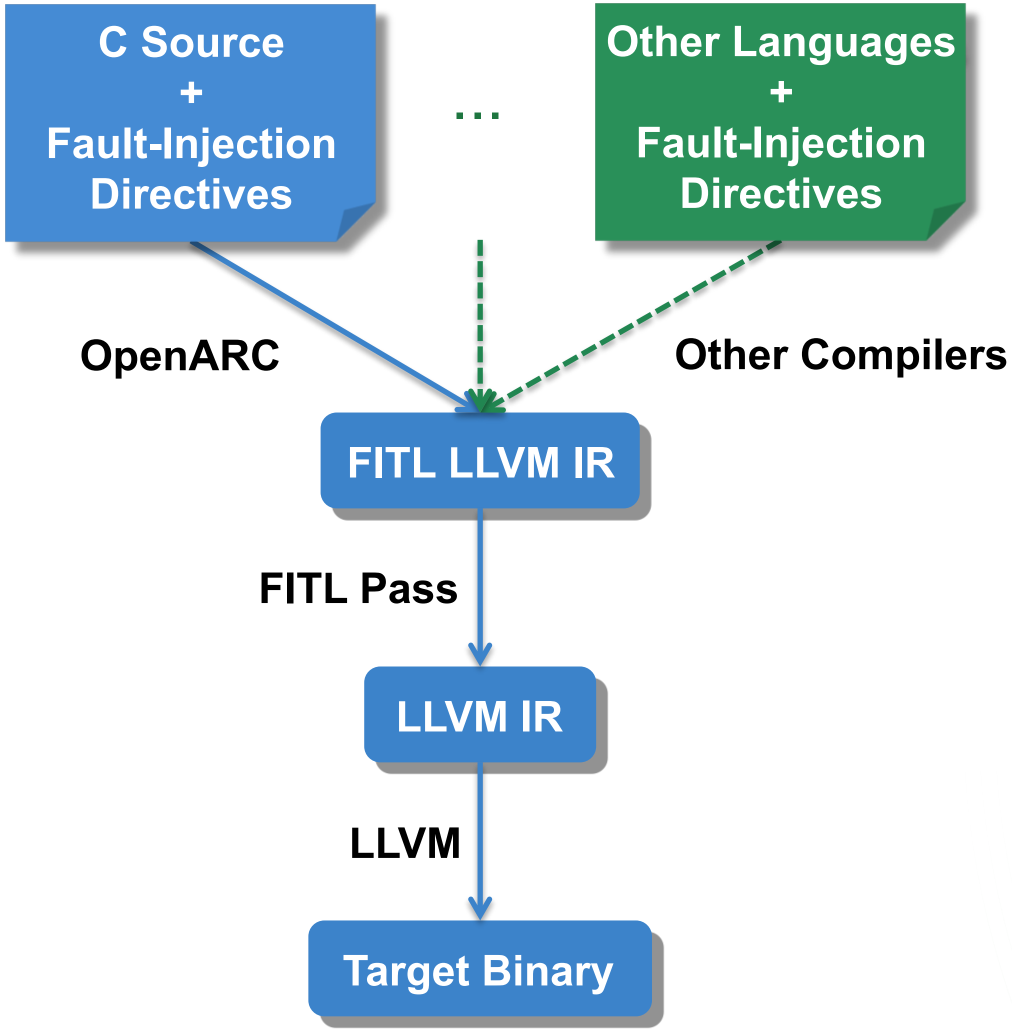


Design Quality vs. Level of Representation





Architecture



C Source
+
Fault-Injection Directives

…

Other Languages
+
Fault-Injection Directives

OpenARC

Other Compilers

FITL LLVM IR

FITL Pass

LLVM IR

LLVM

Target Binary

Contributions

- A set of novel fault-injection pragmas for C
- FITL (Fault-Injection Toolkit for LLVM):
  - FITL LLVM IR: metadata and intrinsics that specify fault injection
  - FITL pass: LLVM pass that lowers FITL LLVM IR to standard LLVM IR
- Abstractions that facilitate the translation of pragmas to FITL
- Fault-injection studies:
  - Includes comparison with source-level fault injector

Fault-Injection Pragmas for C

Fault-Injection Sites

- Fundamental concept at source level and LLVM IR level:
  - Source: sites are defined by directives
  - LLVM IR: site is a basic unit of code inserted to perform fault injection
- Each site is a triple:
  - Execution position: position in code where site is inserted
  - Target: memory or instruction into which a fault might be injected
  - Condition: whether fault is injected
- Sites are defined statically, condition is evaluated dynamically

Pragma: ftinject + ftdata

- Precise specification of a single site targeting memory
- Execution position: exactly where `ftinject` appears: immediately before `printf` call
- Target: `arr[i]`
- Condition: true

```
void print_array(double *arr, int n) {
  for (int i = 0; i < n; ++i) {
    #pragma openarc resilience
    {
      #pragma openarc ftinject ftdata(arr[i:1])
      printf("arr[%d] = %e\n", i, arr[i]);
    }
  }
}
```

Pragma: ftregion + ftdata

- Random selection from region of sites is more realistic
- Execution positions: immediately before every instruction
- Target: `arr[i]`
- Condition: one site is randomly selected

```
void print_array(double *arr, int n) {
  for (int i = 0; i < n; ++i) {
    #pragma openarc resilience
    {
      #pragma openarc ftregion ftdata(arr[i:1])
      printf(“arr[%d] = %e\n”, i, arr[i]);
    }
  }
}
```

Pragma: ftregion + ftkind

- Targets computational units instead of memory
- Execution positions: immediately before every instruction taking a floating point argument
- Targets: the floating pointer arguments
- Condition: one site is randomly selected

```
void print_array(double *arr, int n) {
  for (int i = 0; i < n; ++i) {
    #pragma openarc resilience
    {
      #pragma openarc ftregion ftkind(floating_arg)
      printf(“arr[%d] = %e\n”, i, arr[i]);
    }
  }
}
```

FITL Extensions to LLVM

Basic Block Set

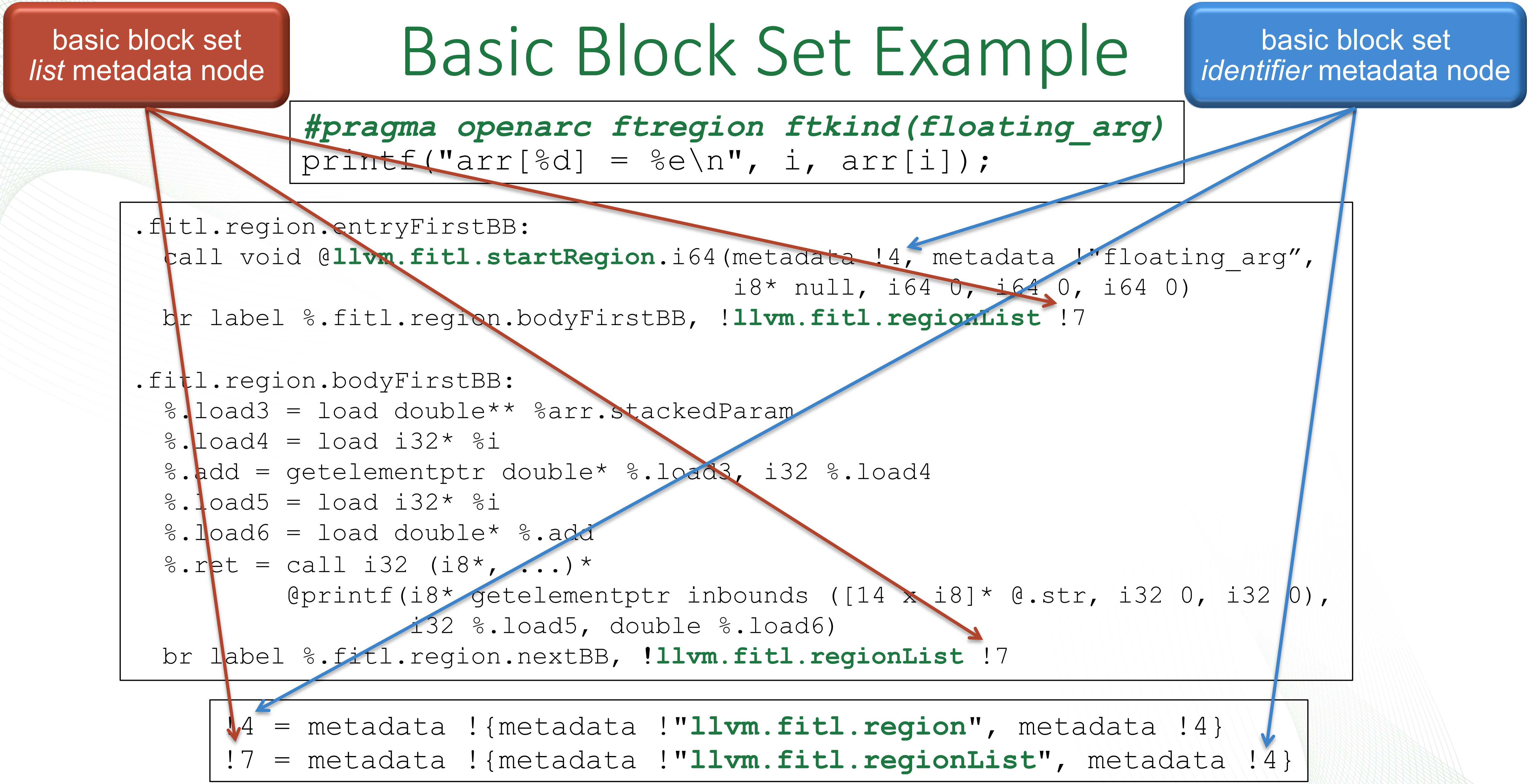
- Each pragma and any attached C block generates a set of LLVM IR instructions
- Compiler front end must communicate to FITL pass which instructions belong to which pragma
- Our choice of granularity is the basic block, so each pragma has a basic block set
- Compiler front end might need to split basic blocks

Entry Intrinsic

- A basic block set *S* can have an entry intrinsic:
  - Called soon after control flow enters *S*
  - Configures functionality within *S*
  - Its args encode clauses of associated pragma

Pragma	Basic Block Set Kind	Entry Intrinsic
`resilience`	`llvm.fitl.domain`	`llvm.fitl.startDomain`
`ftregion`	`llvm.fitl.region`	`llvm.fitl.startRegion`
`ftinject`	`llvm.fitl.desert`	`llvm.fitl.inject`

Basic Block Set Example



basic block set
list metadata node

basic block set
identifier metadata node

```
#pragma openarc ftregion ftkind(floating_arg)
printf("arr[%d] = %e\n", i, arr[i]);
```

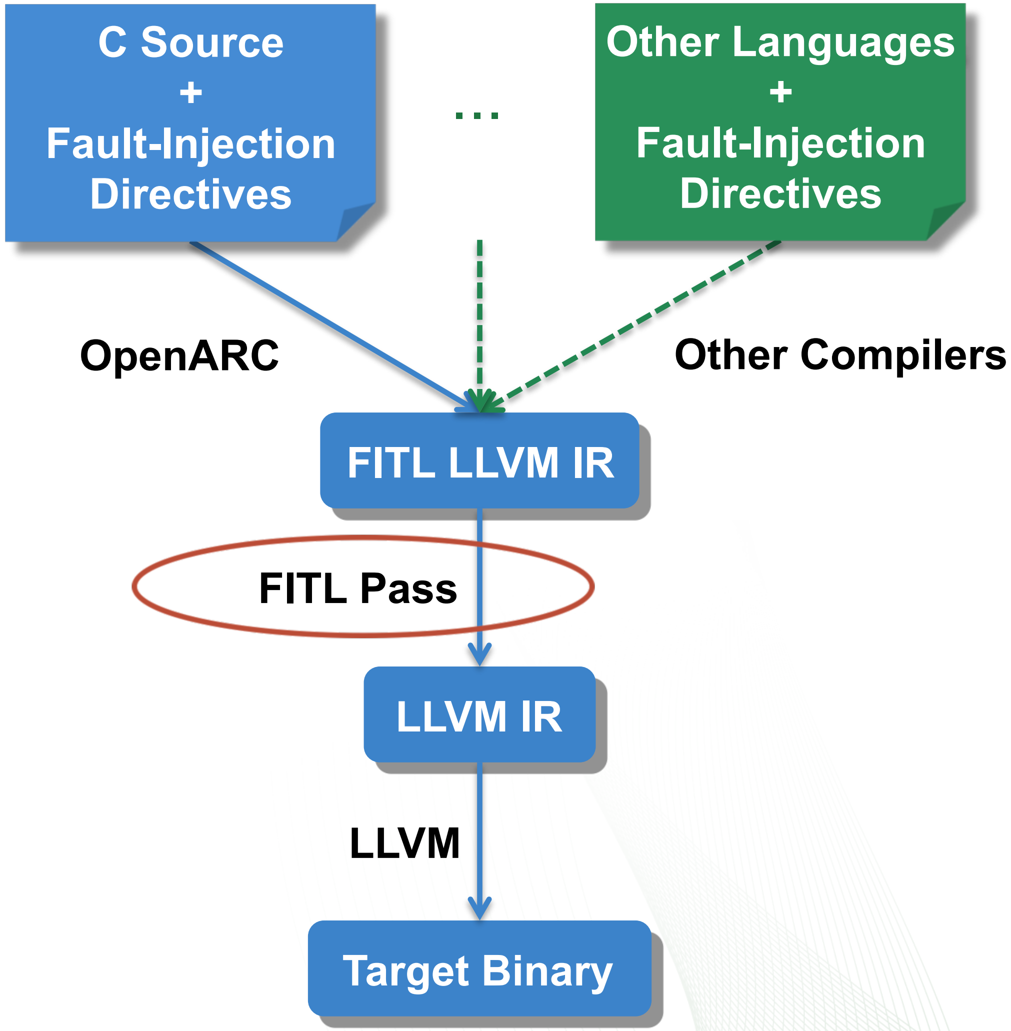
```
.fitl.region.entryFirstBB:
  call void @llvm.fitl.startRegion.i64(metadata !4, metadata !"floating_arg”,
                                       i8* null, i64 0, i64 0, i64 0)
  br label %.fitl.region.bodyFirstBB, !llvm.fitl.regionList !7

.fitl.region.bodyFirstBB:
  %.load3 = load double** %arr.stackedParam
  %.load4 = load i32* %i
  %.add = getelementptr double* %.load3, i32 %.load4
  %.load5 = load i32* %i
  %.load6 = load double* %.add
  %.ret = call i32 (i8*, ...)*
          @printf(i8* getelementptr inbounds ([14 x i8]* @.str, i32 0, i32 0),
                  i32 %.load5, double %.load6)
  br label %.fitl.region.nextBB, !llvm.fitl.regionList !7
```

```
!4 = metadata !{metadata !"llvm.fitl.region", metadata !4}
!7 = metadata !{metadata !"llvm.fitl.regionList", metadata !4}
```

FITL Pass

- Inserts code for fault-injection sites specified by FITL basic block sets and entry intrinsics
- Lowers FITL LLVM IR to standard LLVM IR
- Must be performed before other LLVM passes
- `BasicBlockSet`:
  - LLVM library component we developed
  - Reads basic block set metadata
  - Finds entry intrinsic calls
  - Generally useful for directive-driven programming



Case Studies

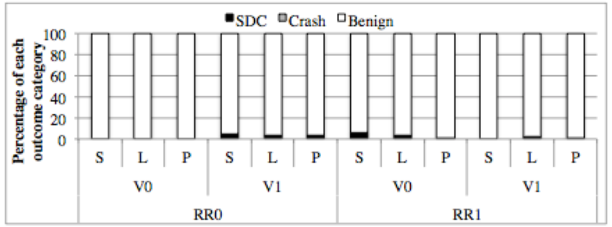
Oak Ridge National Laboratory

Experimental Setup

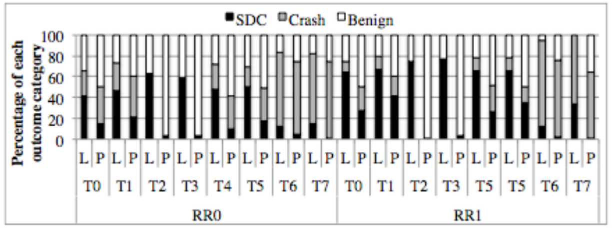
- Studied resilience of four applications:
  - Two kernel benchmarks: JACOBI and MATMUL
  - Two Rodinia benchmarks: BFS and NW
- Each test flips one randomly selected bit, target is either:
  - User data
  - LLVM IR instruction argument of specific type
  - LLVM IR instruction result of specific type
- Each test repeated 100x
- Sequential execution on single CPU on machine that has:
  - Two quad-core Intel Xeon E5520
  - 12GB RAM
  - Scientific Linux Version 6.5

Results

JACOBI

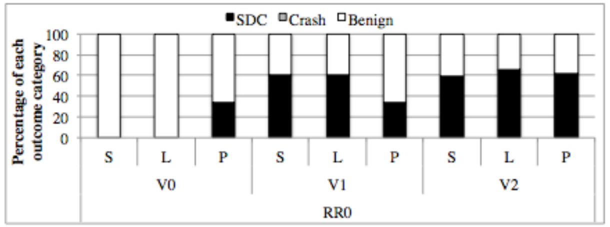


(a) Fault Injections in Memory

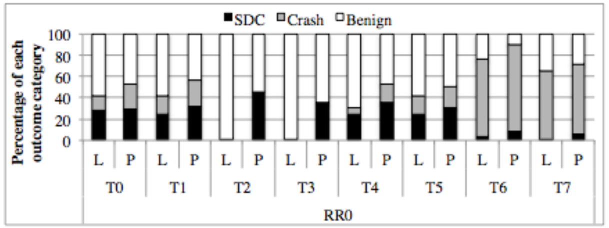


(b) Fault Injections in Computational Units

MATMUL

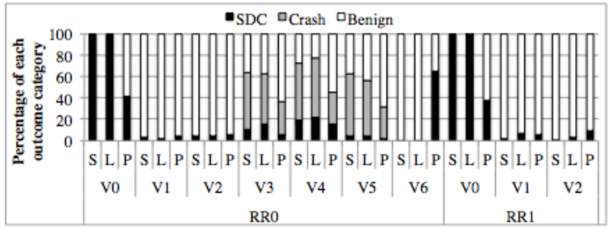


(a) Fault Injections in Memory

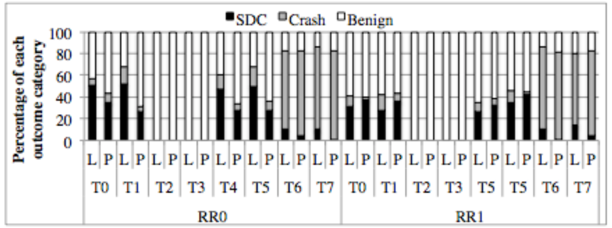


(b) Fault Injections in Computational Units

Rodinia BFS

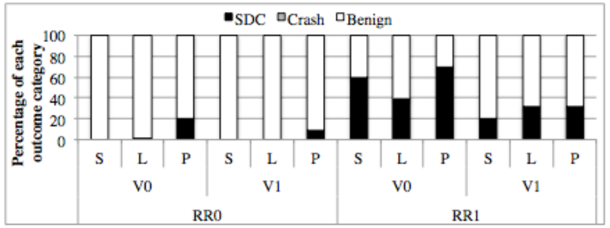


(a) Fault Injections in Memory

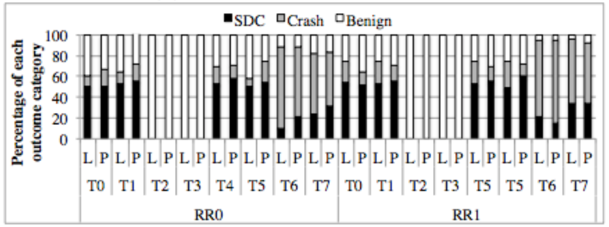


(b) Fault Injections in Computational Units

Rodinia NW



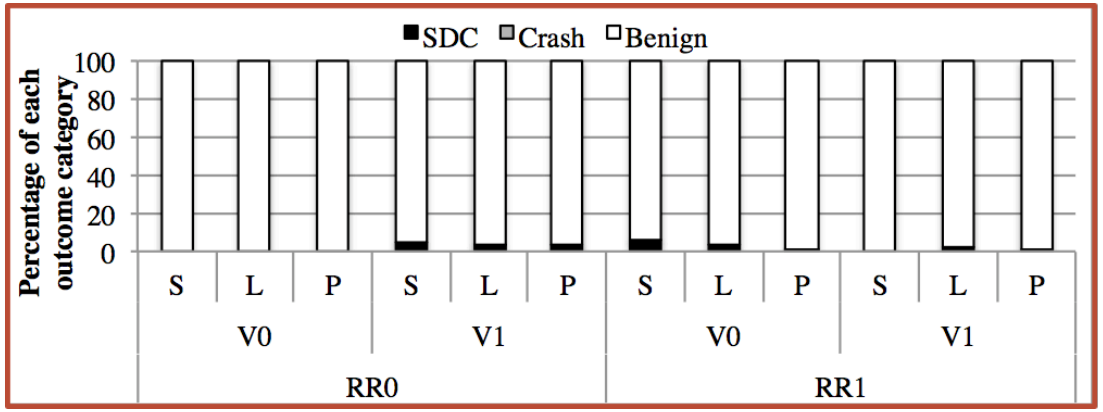
(a) Fault Injections in Memory



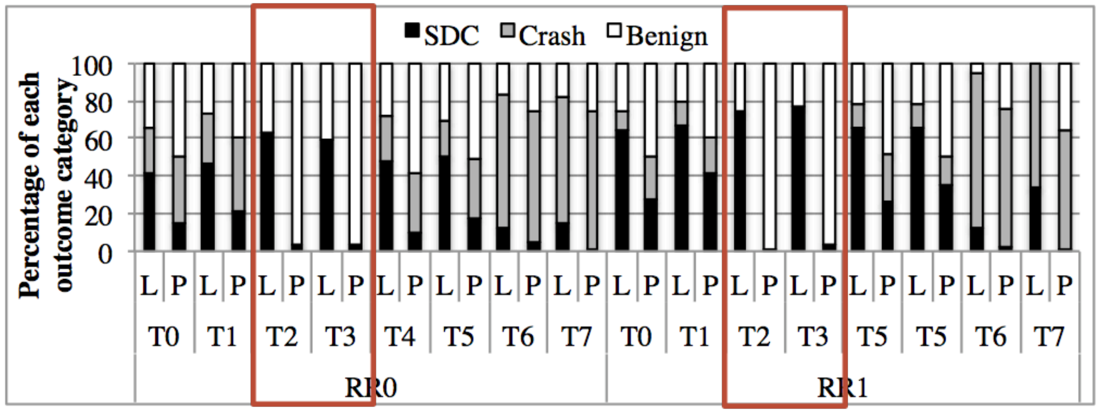
(b) Fault Injections in Computational Units

JACOBI

Symbol	Description
S	Non-profiled source-level fault injections by OpenARC
L	Non-profiled LLVM-IR-level fault injections by FITL
P	Profiled LLVM-IR-level fault injections by FITL
T0	Target integer arguments of LLVM instructions
T1	Target integer results of LLVM instructions
T2	Target float arguments of LLVM instructions
T3	Target float results of LLVM instructions
T4	Target arguments of arithmetic LLVM instructions
T5	Target results of arithmetic LLVM instructions
T6	Target pointer arguments of LLVM instructions
T7	Target pointer results of LLVM instructions
RRn	Code block annotated with a resilience pragma
Vn	User variable where faults are injected



(a) Fault Injections in Memory

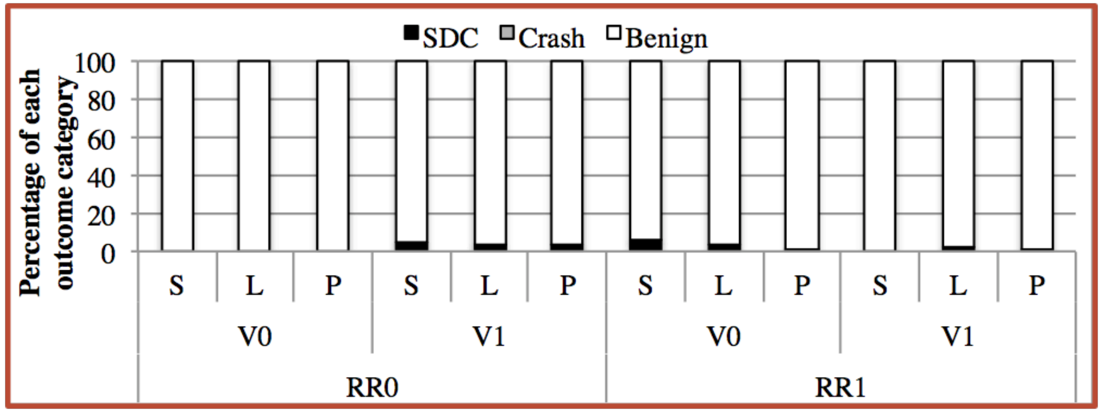


(b) Fault Injections in Computational Units

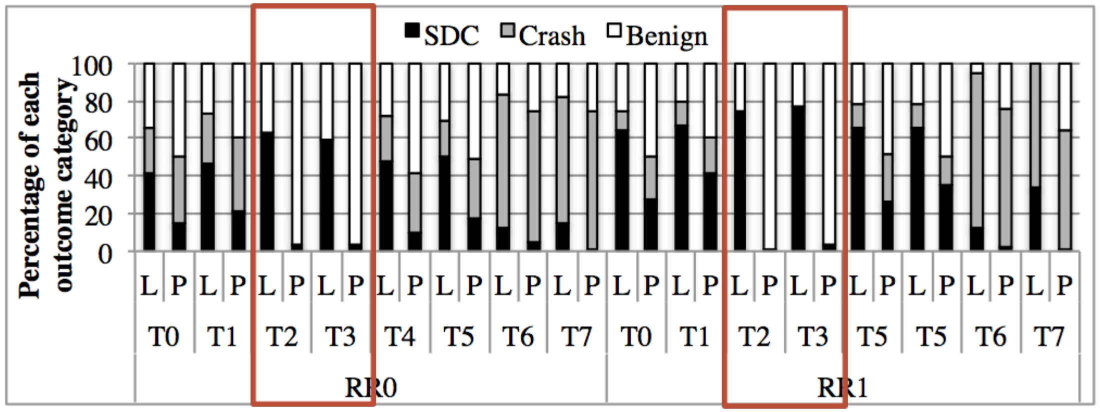
fault injection into floating point

JACOBI

- Faults in float memory:
  - No crashes:
    - Data doesn't affect control flow
    - Data doesn't include pointers
  - Few SDCs:
    - Data overwritten every iteration
    - Roundoff error
- Faults in float computation:
  - No crashes (same reasons)
  - Few SDCs for profiled (P)
  - Many SDCs for non-profiled (L)



(a) Fault Injections in Memory



(b) Fault Injections in Computational Units

Acknowledgements

- Contributors and Sponsors
  - Future Technologies Group: http://ft.ornl.gov
  - US Department of Energy Office of Science
    - DOE ARES Project: http://ft.ornl.gov/research/ares

Oak Ridge National Laboratory