UCLA Samueli Computer Science

SE4~~AI~~

*Lessons Learned from Designing SE Methods for **Big Data** and **HW Heterogeneity***

Miryung Kim

Professor and Vice Chair of Graduate Studies

UCLA Computer Science

SEMLA June 9th 2023, Montreal, Canada

Software Engineering Analysis Lab at UCLA

Code Mining, Debugging and Refactoring for Java

Debugging and Testing Tools for Big Data

Systems and Runtimes

Developer Tools for Heterogeneous Computing

AI4SE

AI for Software Engineering

SE4AI

Software Engineering for Enabling AI

A new wave of SE tools for data and compute-intensive

Take-away: Context

There are huge challenges for making automated SE tractable in the data-intensive and compute-intensive domain.

- the scale of large **input**
- long invocation **latency**
- performance **overhead**
- **layers** and layers

Take-away: Lessons Learned

Abstraction is useful for speed.

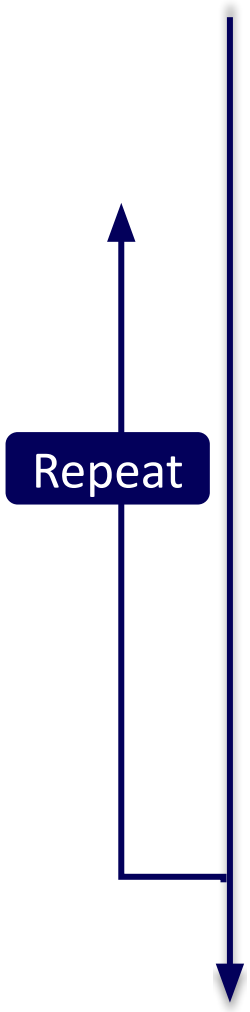
Injecting debuggability requires re-design.

Systems-level optimizations are necessary.

Domain-specialization is necessary and we cannot wait for a large corpus to exist first.

Part 1.
SE Tools for
Big Data Analytics

Big Data Analytics Lifecycle



Develop

1

- Design the application locally

Test

2

- Test locally on a sample data

Run

3

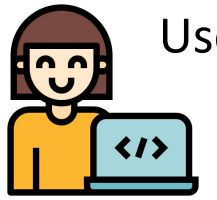
- Execute on the cloud, hoping that it would work

Debug

4

- Several hours later, the job crashes

Apache Spark 101



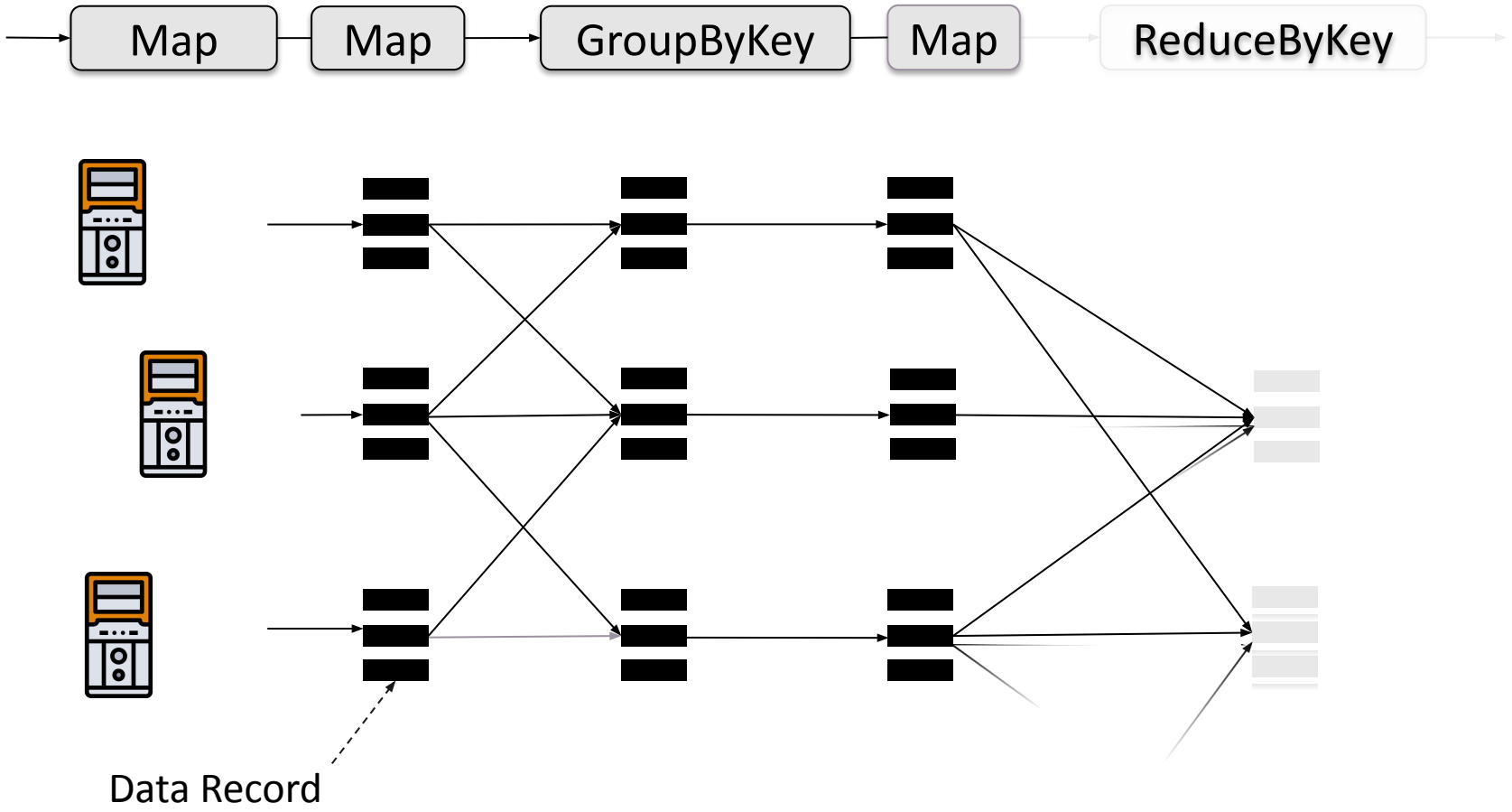
User Machine

Distributed Cluster

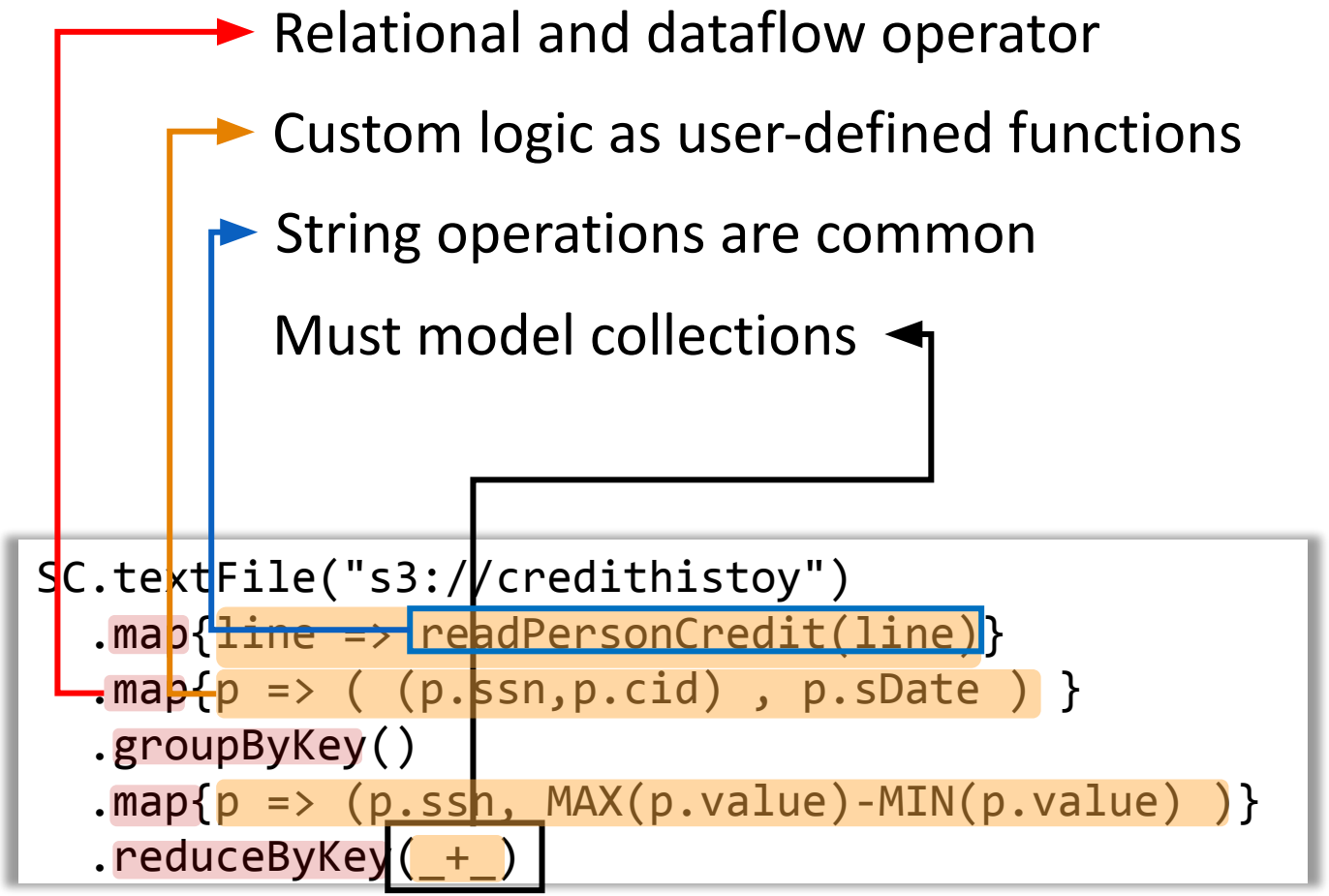
```
SC.textFile("s3://credit history")
   .map{line => readPersonCredit(line)}
```

Dataflow Operator

User Defined Function (UDF)



Apache Spark 101



Challenge: Complexity of Apache Spark

- Dataflow operators are built on 100Ks lines of framework's code with complex systems artifacts; therefore **symbolically executing them is infeasible**.

Internal implementation of JOIN

```
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))] =
  this.cogroup(other, partitioner).flatMapValues(
     pair =>
         for (v <- pair._1.iterator; w <- pair._2.iterator) yield (v, w) )
}
override def compute() = {
  val rddIterators = new ArrayBuffer[(Iterator[Product2[K, Any]], Int)]
  for ((dep, depNum) <- split.deps.zipWithIndex) dep match {
    case NarrowCoGroupSplitDep(rdd, _, itsSplit) => . . .
```

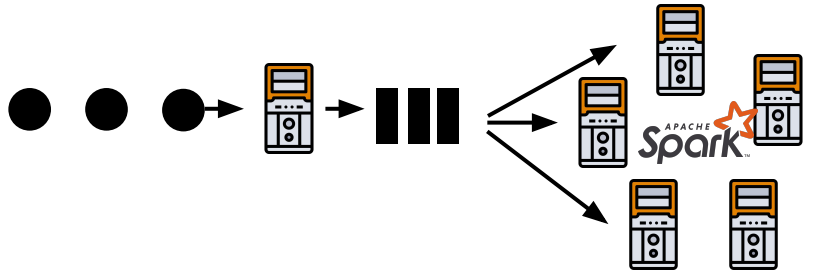
Higher-order function | Query Optimizer | Concurrency Constructs

Data Partitioning | MPI

Challenge: Test Input Generation

Generate Test Data

- Use symbolic execution to automatically generate test data.



Limitation

- Symbolic execution works best to produce unit tests for small code snippets.

BigTest Approach [FSE 2019]

Inputs:  Big Data Application

Output:  Synthetic Test Data

```
SC.textFile("s3://credithistoy")
  .map{line => readPersonCredit(line)}
  .map{p => ( (p.ssn,p.cid) , p.sDate ) }
  .groupByKey()
  .map{p => (p.ssn, MAX(p.value)-MIN(p.value) )}
  .reduceByKey(_+_)
```

2 Symbolically execute UDFs

Path Constraint:
PI.split(“,”).length >1 ∧
AA.split(“,”)[0] = PI.split(“,”)[0] ∧
AA.split(“,”)[3].isInteger ∧ …
Effect:
AA.split(“,”)[0] + AA.split(“,”)[1] +
AA.split(“,”)[2]

1 Dataflow operators in first order logic

Reduce:
∃tR,tL: cR ∈ CR ∧ cL ∈ CL ∧ cR(tR) ∧ tR,key = tL,key ∧cL(tL)
Map:
∃tR,tL: cR ∈ CR ∧ cL ∈
..

4 Generate test data using constraint solvers

3 Join dataflow operator logic with the symbolic representation of UDFs

BigTest Evaluation

34%

Code Coverage Improvement

BigTest improves JDU path coverage by 34% against the entire dataset.

2X

Fault Detection

BigTest detects 2X more faults than Sedge because it models the internal semantics of UDFs with the specifications of dataflow operators.

10^5 to 10^8X

Test Data Size Reduction

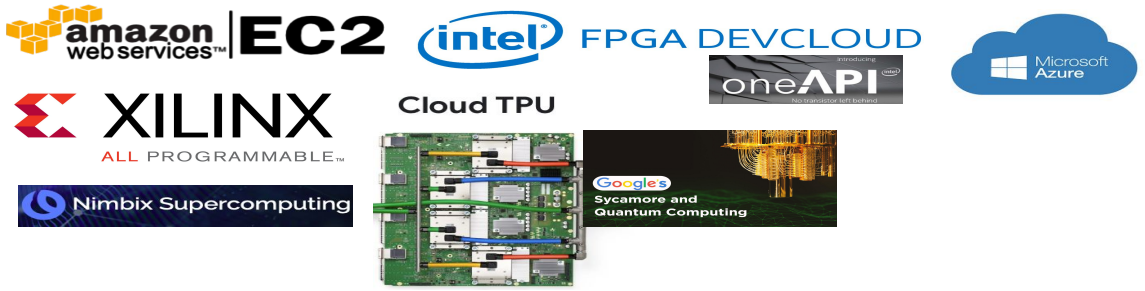
Compare to the entire data set, Big Test archives more code coverage with smaller data

194X

Speed Up

It reduces testing time by 194X.

Part 2.
SE Tools for
Heterogeneous Computing



Hardware accelerators are widely available

amazon web services™ | EC2

intel® FPGA DEVCLOUD

Introducing oneAPI
No transistor left behind

Microsoft Azure

XILINX
ALL PROGRAMMABLE™

Cloud TPU

Nimbix Supercomputing

Field programmable gate array (FPGA)

Programmable logics, interconnects, and customizable building blocks

Catapult – Bing search with FPGA-enabled servers
50% throughput increase and 25% latency reduction.

Difficult to programs in RTL languages

Microsoft

Developer workflow with High Level Synthesis

```
int KNN()
  ...
// Calculate distance
   for (i = 0 to
number){
   dist[i] =
l2norm(data[i], dim);
   }
//Top 1 nearest
neighbor
    ...
}
```

7X speed up on FPGA

1. Performance profiling

2. Kernel function identification in C

3. Manual **rewriting** from C to HLS-C

4. **Differential testing** with input samples (RTL simulation vs. C execution)

5. Iterative optimization

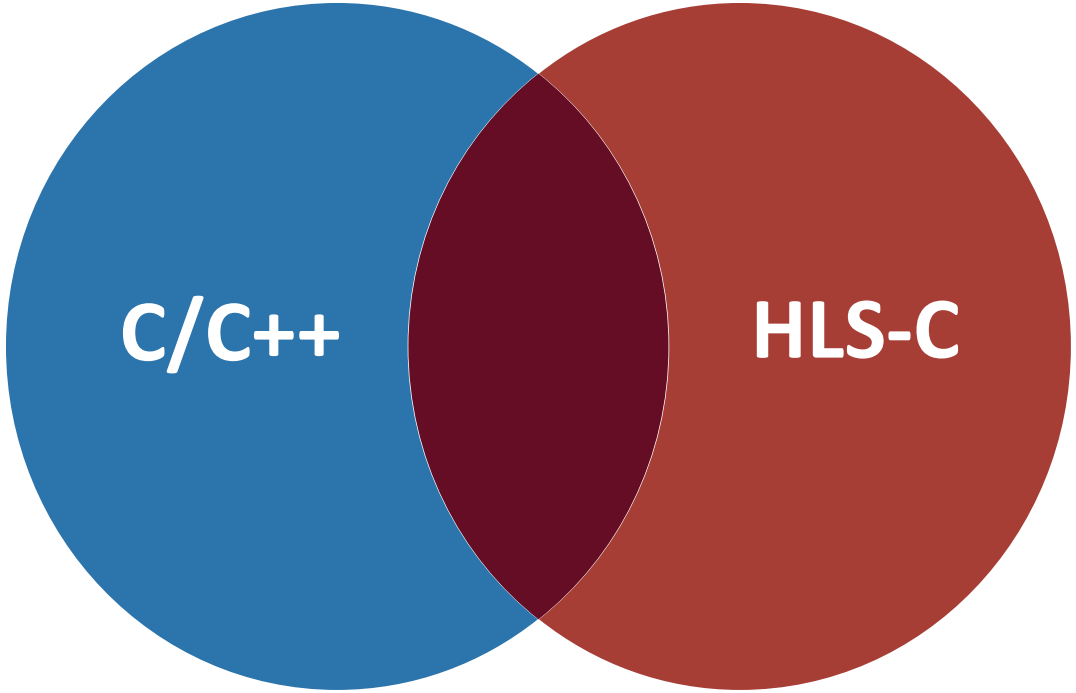
HLS compilation to RTL
6 minutes

CPU-FPGA co-simulation
8 minutes

Repeat

FPGA synthesis in 2.5 hours

HLS tools are not easy to use for SW developers



Manual rewriting for synthesizability and optimization

No developer tools for code translation

- Resource **finitization**
- Hardware expertise and **pragmas (directives)** for optimization
- Partitioning, parallelization, pipelining, etc.

HLS-C requires specifying *bitwidth* for each type

```
float vecdot(
    float a[],
    float b[],
    int n) {
    for (int i = 0; i < n;
i++)
        sum += a[i] * b[i];
    return sum;
}
```

C Program

```
float vecdot(
    float a[],
    float b[],
    fpga_int<7> n) {
    for (fpga_int<7> i = 0;
i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

HLS-C Program

HLS-C uses a *custom* floating point type

```
float vecdot(
    float a[],
    float b[],
   fpga_int<7> n) {
    for (fpga_int<7> i = 0; i
< n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

C Program

```
fpga_float<8,15> vecdot(
    fpga_float<8,15> a[],
    fpga_float<8,15> b[],
   fpga_int<7> n) {
    for (fpga_int<7> i = 0; i < n;
i++)
        sum += a[i] * b[i];
    return sum;
}
```

HLS-C Program

HLS-C requires *finitizing* resources

HLS compile error

```
struct Node {
     Node *left, *right;
     int val; };
void init(Node **root) {
     *root = (Node
*)malloc(sizeof(Node)); }
void delete_tree(Node *root) {...
     free(root); }
void traverse(Node *curr) {
     if (curr == NULL) return;
     int ret = visit(curr->val);
     traverse(curr->left);
     traverse(curr->right);
}
```

C Program

```
Node Node_arr[NODE_ARR_SIZE];
struct Node {
     Node *left, *right;
int val; };

void delete_tree(Node_ptr root)
{...
     node_free(root); }
void traverse_converted(Node_ptr
curr) {
    stack<context> s(STACK_SIZE);
    while (!s.empty()) {
       ...}}
```

HLS-C Program

Divergence errors between CPU and FPGA

```
int main(int argc, char
*argv[]){
int data[] =
  gradient(argv[1]);
  int sum;
  float th = argv[2];
  int size = data.size();
  accumulate(data[size]);
  for(i = 0 to size){
    data[i] /= sum;
    if(data[i] > th)
       discard;
  }
}
```

Host Code

```
int accumulate(int data[size]){
  typedef ap_uint<8> bit8;
  #define max M;
  bit8 sum = 0;
  bit8 data_fpga[M];
  for(i = 0 to M){
    data_fpga[i]=(bit8)data[i];
  }
  SUM_LOOP for(i = 0 to M){
    #pragma HLS unroll factor=2
    sum += data_fpga[i];
  }
  return sum;
}
```

Kernel Code

Input	CPU	FPGA
[1,1,1,253]	no errors	div/0 in host
[2,1,1,253]	257	1

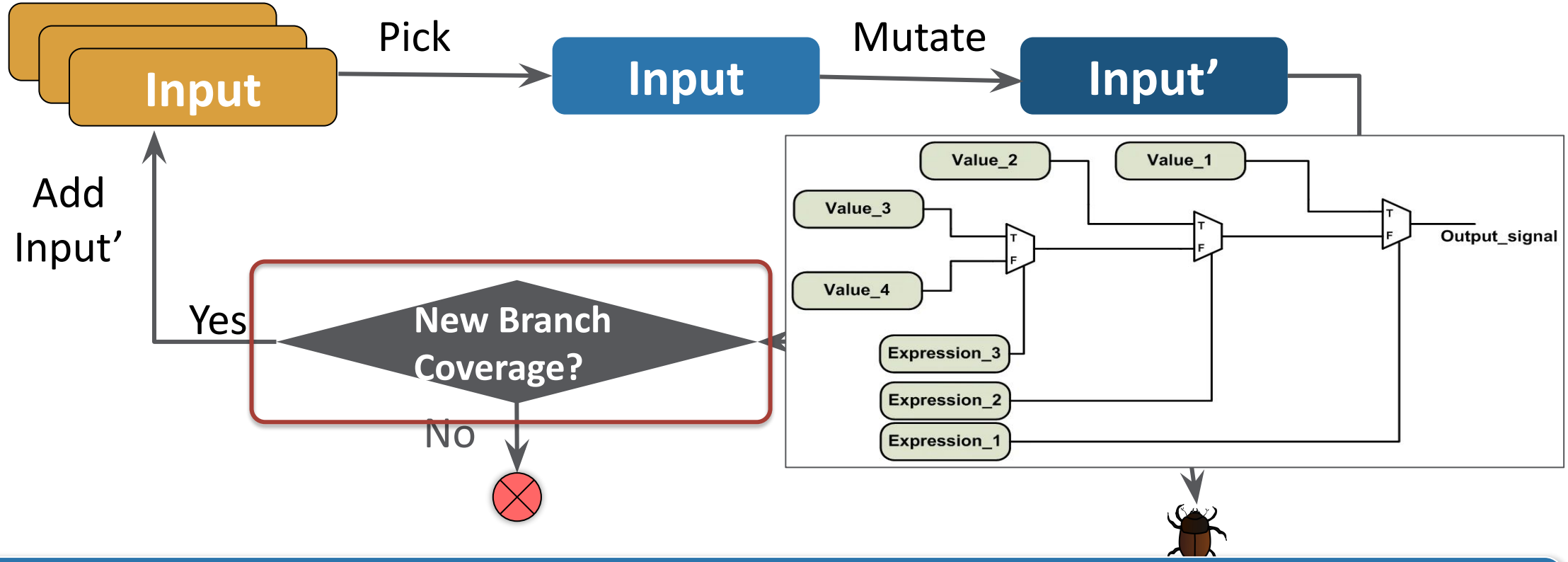
AFL running time for finding errors

Table 1: Examples of Behavior Divergence Between CPU and FPGA

ID	Description	Time
894069 [57]	Segmentation fault when allocating a big array `int x[1920][1080]` on FPGA yet no error on CPU	8.5h
595225 [58]	Different outcome caused by HLS `dataflow` directive	47.7h
438446 [59]	Different outcome caused by FPGA fetching incorrect `struct vector training_set[MAXSZ]`	10.6h
754676 [60]	Different outcome caused by bitwidth `typedef ap_fixed<25,1,AP_RND> s25f24_type`	2.3h
785019 [61]	Getting all zeros when shifting an array caused by `#pragma HLS RESET`	3.1h
907213 [62]	Undecided output when overwriting a same variable within the loop yet no error on CPU	79.4h
1166264 [63]	`EMFILE` error when loading 2048 files with `#pragma HLS ARRAY_PARTITION` yet no error on CPU	11.7h
1126600 [64]	The 25-tap FIR filter bypassess some input multiplications with `#pragma HLS PIPELINE`	93.2h

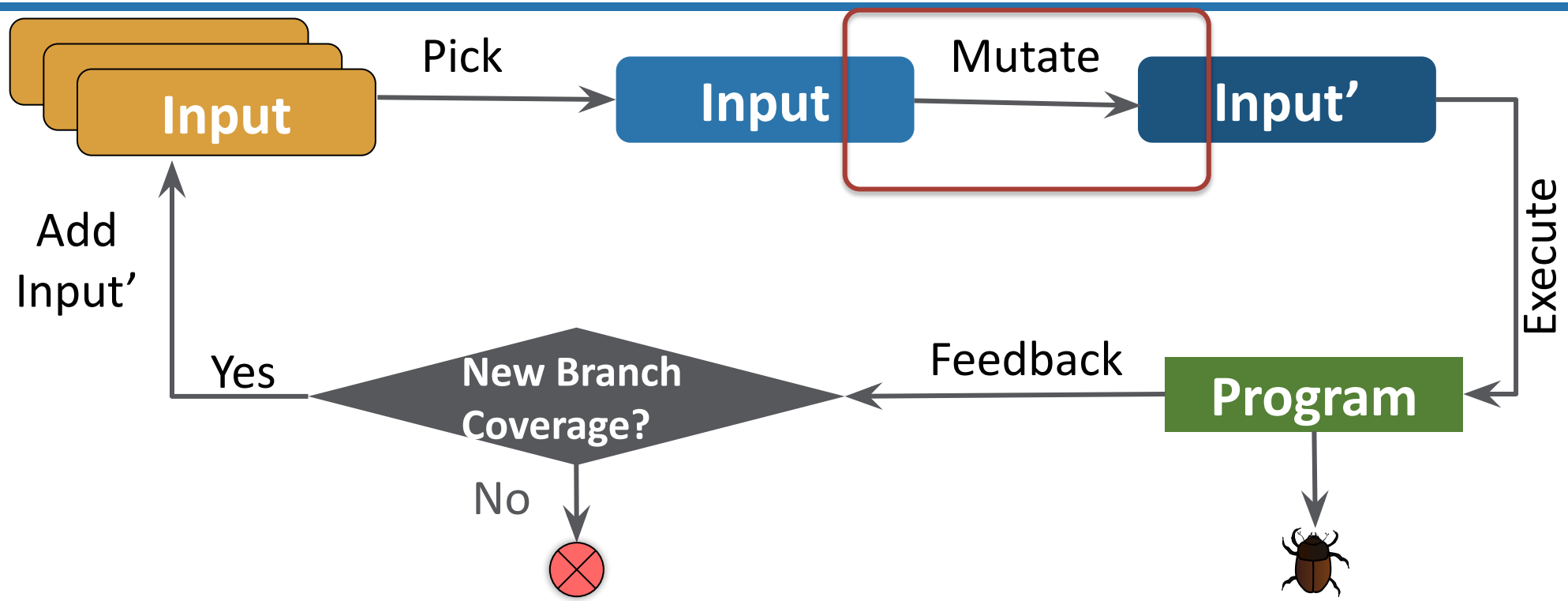
AFL: American Fuzzy Lop (a well known fuzz testing framework)

Challenge 1: lack of guidance in HW



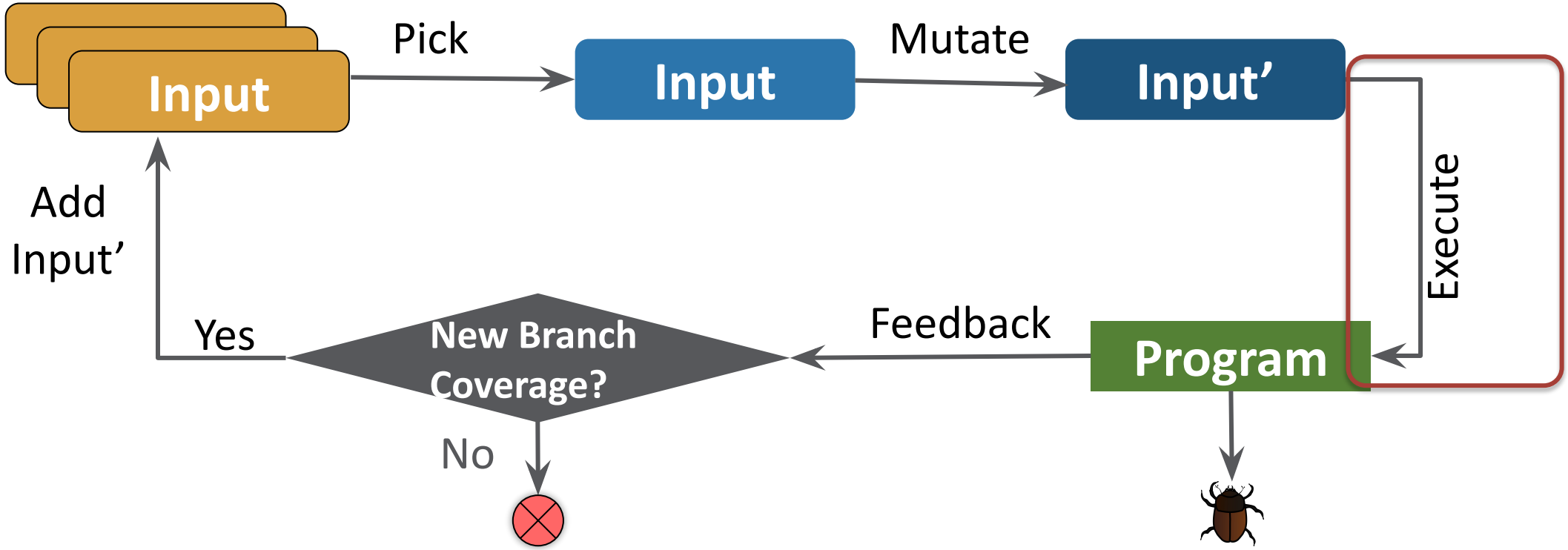
Branch coverage is not meaningful in HW

Challenge 2: lack of effective mutations



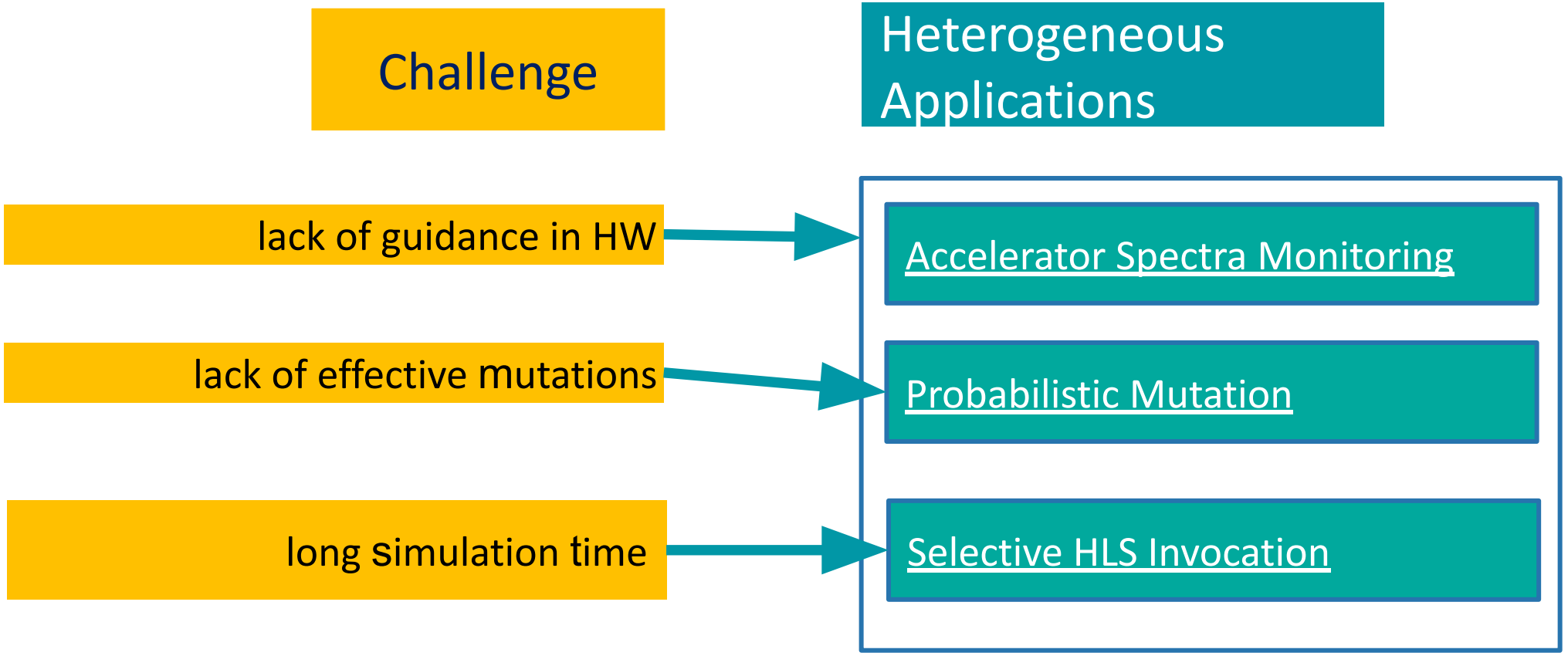
Input mutations must stretch HW behavior in terms of finitized resource usages to induce errors

Challenge 3: long simulation time



Fuzzing assumes the program under test can execute quickly in the order of milliseconds.

HeteroFuzz Approach (FSE 2021)



Accelerator spectra monitoring

```
int accumulate(int data[size]){
  typedef ap_uint<8> bit8;
  #define max M;
  bit8 sum = 0;
  bit8 data_fpga[M];
  for(i = 0 to M){
    data_fpga[i]=(bit8)data[i];
  }
  SUM_LOOP for(i = 0 to M){
    #pragma HLS unroll factor=2
    sum += data_fpga[i];
  }
  return sum;
}
```

Kernel Code

Static analysis of HLS pragmas

```
int main(int argc, char
*argv[]){
int data[] =
  gradient(argv[1]);
  int sum;
  float th = argv[2];
  int size = data.size();
  accumulate(data[size]);
  for(i = 0 to size){
    data[i] /= sum;
    if(data[i] > th)
       discard;
  }
}
```

Host Code

Inject accelerator specific monitors

Fuzzing Guidance

Kernel input: [1,1,1,9]

Accelerator Feedback

- Data_fpga: [1,9]
- Sum: [2,12]
- Accessed offsets: [0,1,2,3]
- loop iterations: 2

Host Feedback

The activated branches

HeteroFuzz evaluation

754X

Speed up

with three-pronged optimizations in finding divergence errors

57%

Effectiveness

divergence-inducing inputs with accelerator spectra monitoring

17.5X

Speed up

with dynamic probabilistic mutation

8.8X

Efficiency

with selective HLS invocation

What lessons have we learned by adapting SE methods to the **data-intensive** and **compute-intensive domain**?

Project	Key Idea
BigTest: **Symbolic Execution** [FSE 2019]	Abstraction to tame complexity: combining UDF symbolic execution with dataflow and relational operator semantics
BigFuzz: **Fuzz Testing** [ASE 2020]	Abstraction to increase speed: source-level API rewriting for dataflow and relational APIs
BigDebug: **Interactive Debugger** [ICSE 2016]	Rewriting runtime to add inspection visibility into opaque and complex data processing without much overhead
PerfDebug: **Performance** Latency **Debugging** [SoCC 2019]	Rewriting runtime to add inspection capability for fine-grained latency tracking
Titian: Data **Provenance** [VLDB 2015]	Distributed, optimized backward join via partition-id tracking
BigSift: **Delta Debugging** [SoCC 2017]	Systems-level optimizations for memoizing similar executions and reducing the scope of inputs to re-execute
FlowDebug: **Taint Analysis** using Influence Function [SoCC 2020]	Winnowing out answers for large-scale aggregation operators
OptDebug: **Spectra-based Debugging**/ Operation Tainting [SoCC 2021]	Winnowing out answers via operation-level tainting

Lesson 1: **Abstraction** is useful for increasing speed.

Project	Key Idea
BigFuzz: Fuzz Testing [ASE 2020]	Abstraction to increase **speed**: source-level API rewriting for dataflow and relational APIs
BigTest: Symbolic Execution [FSE 2019]	Abstraction to tame **complexity**: combining UDF symbolic execution with dataflow and relational operator semantics

Broken assumptions

- A program runs fast to allow fuzzing. 😢
- Code is too big for symbolic execution. 😊

Lesson 2: **Injecting debuggability** and **traceability** requires re-design.

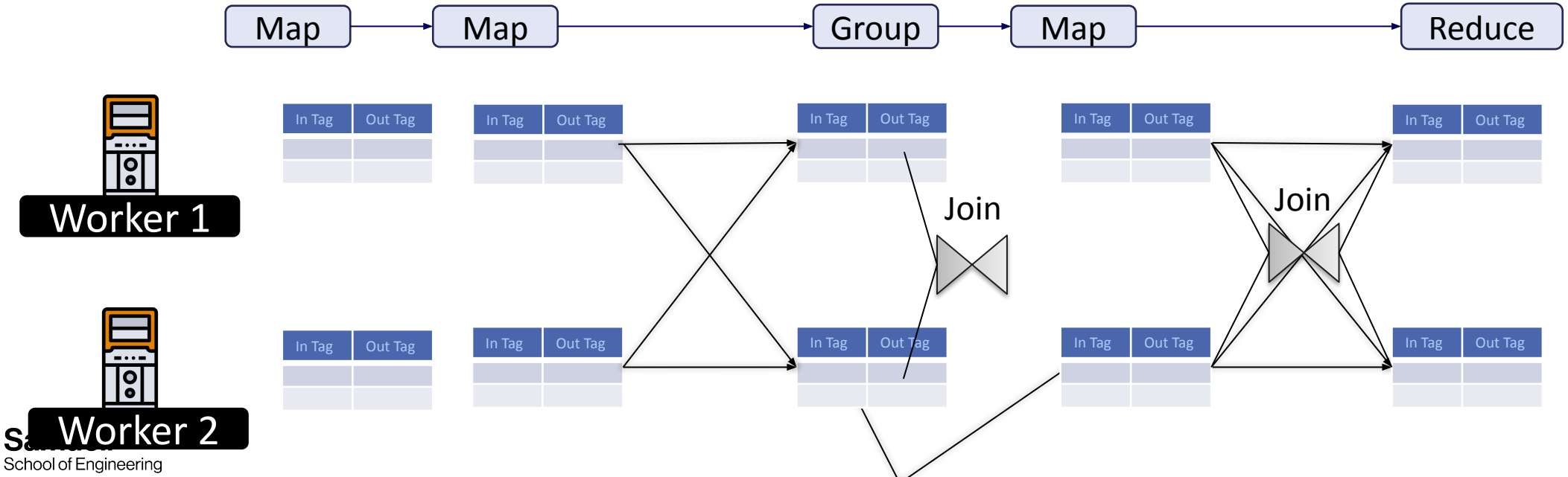
Project	Key Idea
BigDebug: Interactive Debugging [ICSE 2016]	**Rewriting runtime** to add inspection visibility into opaque and complex data processing without much overhead
PerfDebug: Performance Latency Debugging [SoCC 2019]	**Rewriting runtime** to add inspection capability for fine-grained latency tracking

Broken assumptions

- Injecting traceability into runtimes adds intolerable 😊 overhead. Replay debugging is too slow.

Lesson 3: **Systems-level optimizations** are necessary

Project	Key Idea
Titian: Data Provenance [VLDB 2015]	Distributed, optimized backward join via partition-id tracking
BigSift: Delta Debugging [SoCC 2017]	**Systems-level optimizations** for memoizing similar executions and pushing oracle evaluation to earlier computation stages



Lesson 4: **Domain-specialization is** necessary, yet no large corpus exists for a new domain.

Project	Key Idea
HeteroRefactor [ICSE 2020]	Optimization of bitwidths and FP precision: accuracy and performance tradeoffs
HeteroFuzz [FSE 2021]	Test input generation that accounts for HW accelerator behavior
HeteroGen [ASPLOS 2022]	Automated program repair for HLS rewriting

Broken assumptions

- A program runs fast to allow fuzzing. 😢
- A compiler runs fast to allow iterative program repairs. 😢
- Many examples exist to allow code pattern mining. 😢

What challenges and opportunities exist in this data and compute-intensive domain?

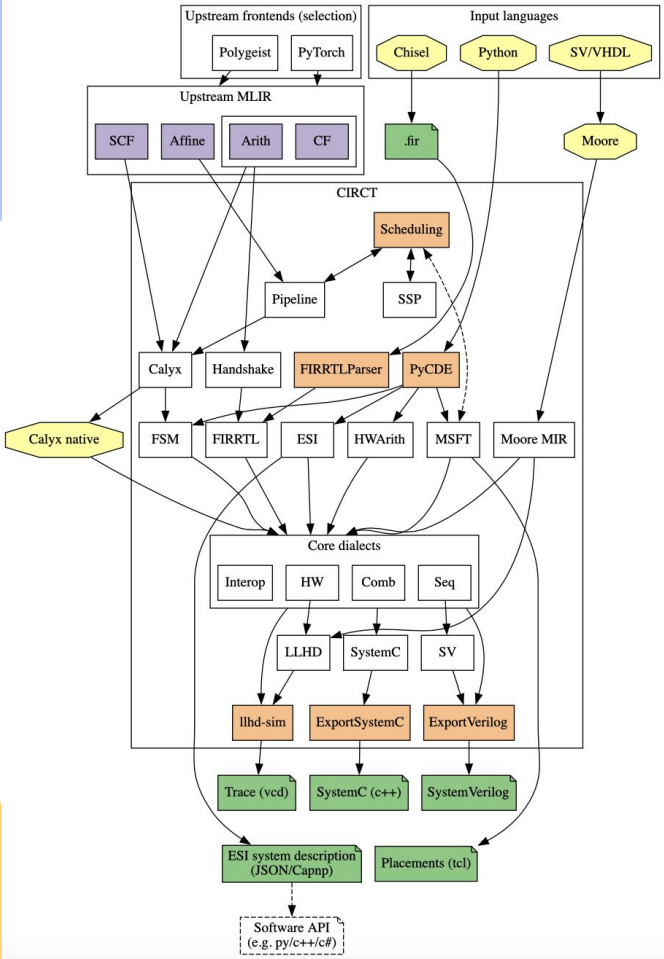
Future Work: Challenges and Opportunities

Challenges	Slow Program Execution
Opportunities	HW acceleration for test generation • invocation • mutation • in-kernel probes

From targeting heterogeneity to
leveraging heterogeneity

Future Work: Challenges and Opportunities

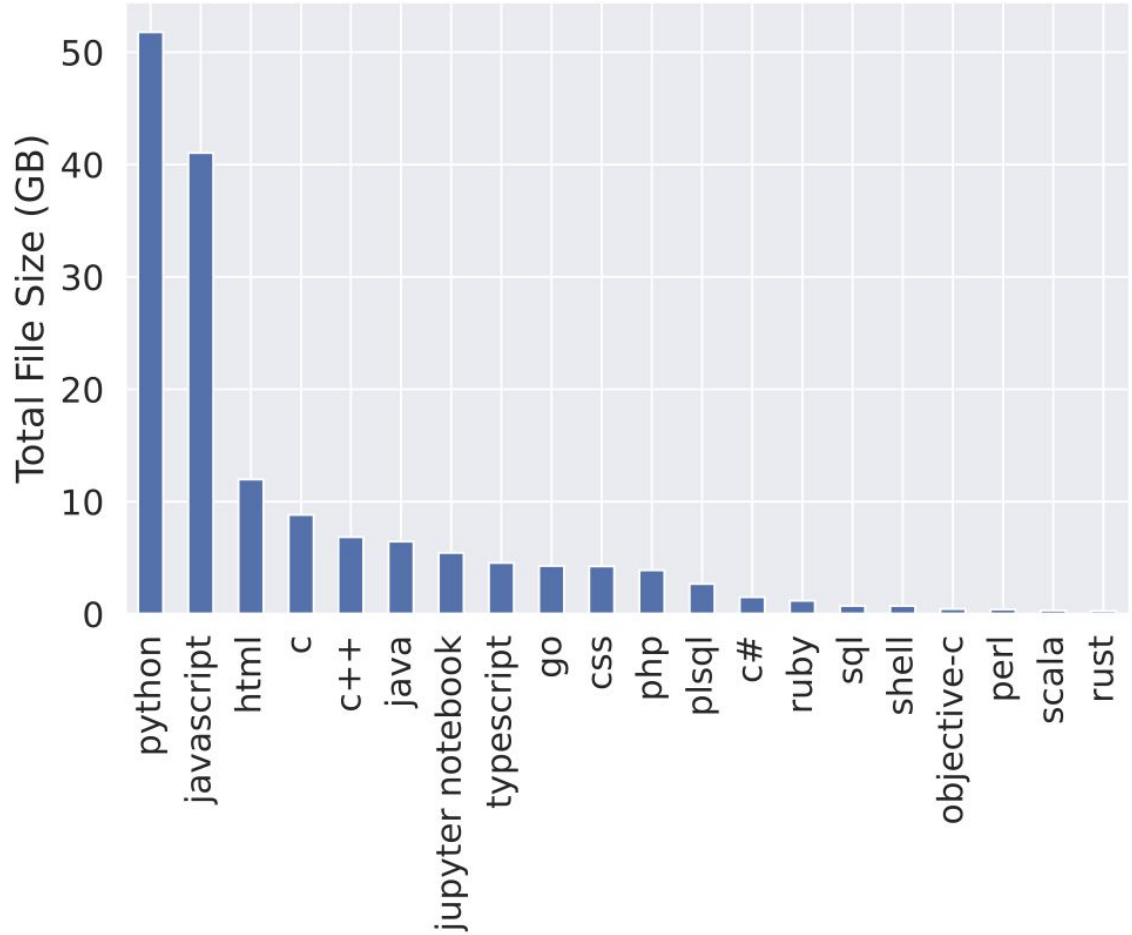
Challenges	Layers and Extensions
Opportunities	Test generation for multi-layer compiler extension Isolating an infection chain across multi layer



Targeting a system of systems
not a standalone system

Future Work: Challenges and Opportunities

Challenges Domain specialization
in the absence of a large corpus



Corpus statistics for Incoder [ICLR 2023]

Future Work: Challenges and Opportunities

Challenges	Domain specialization in the absence of a large corpus
Opportunities	Human-in-the-loop synthesis

- custom mutation
- custom feedback

Fuzzing

- custom repair exploration
- custom fitness function

Repair

- custom code patterns

Search

Active incorporation of human feedback, while reducing inspection effort

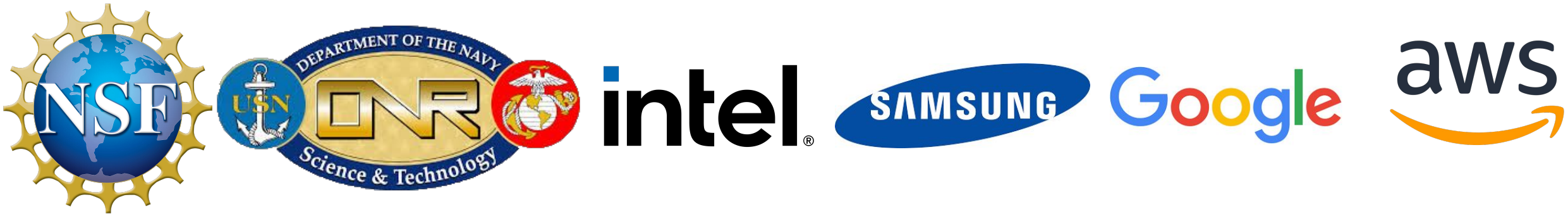
Summary

In the era of AI, we must re-imagine SE methods to account for the importance of **big data** and increasing **hardware heterogeneity.**

There are huge challenges for making automated SE tractable in the data-intensive and compute-intensive domain: the scale of large **input,** long invocation **latency,** performance **overhead,** and **layers** and layers.

Thank you!

Thanks to Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Jason Lau, Aishwarya Sivaraman, Jason Cong, Harry Xu, Hongbo Rong, Rohan Padhye, Jason Teoh, Fabrice Harel-Canada, Yifan Qiao, Haoran Ma



https://github.com/ucla-seal/

Developer Tools for Heterogeneous Computing

HeteroGen: Automated Transpilation and Repair from C to HLS-C (ASPLOS 2022)
HeteroFuzz: Test Input Generation for Differential Testing CPU vs. FPGA (FSE 2021)
HeteroRefactor: Refactoring for Heterogeneous Applications with FPGA (ICSE 2020)
QDiff: Automated Testing of Quantum Software Stacks (ASE 2021)

Debugging and Testing Tools for Big Data

BigTest: Symbolic-Execution based Test Input Generation, (FSE 2019)
BigFuzz: Fuzz Testing, ASE 2020
BigDebug: Interactive Debugger (ICSE 2016)
BigSift: Automated Delta Debugging (SoCC 2017)
PerfDebug: Performance Debugging (SoCC 2019)
FlowDebug: Dynamic Taint Analysis with Influence Function (SoCC 2020)
OptDebug: Operational-Level Taint Analysis Spectra Debugging SoCC 2021