2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) | 979-8-3503-2496-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-FOSE59343.2023.00006

# Software Engineering for Data Intensive Scalable Computing and Heterogeneous Computing

Miryung Kim
*UCLA*
Los Angeles, USA
miryung@cs.ucla.edu

***Abstract*—With the development of big data, machine learning, and AI, existing software engineering techniques must be re-imagined to provide the productivity gains that developers desire. Furthermore, specialized hardware accelerators like GPUs or FPGAs have become a prominent part of the current computing landscape. However, developing heterogeneous applications is limited to a small subset of programmers with specialized hardware knowledge. To improve productivity and performance for data-intensive and compute-intensive development, now is the time that the software engineering community should design new waves of refactoring, testing, and debugging tools for big data analytics and heterogeneous application development.**

**In this paper, we overview software development challenges in this new data-intensive scalable computing and heterogeneous computing domain. We describe examples of automated software engineering (debugging, testing, and refactoring) techniques that target this data and compute intensive domain and share lessons learned from building these techniques.**

***Index Terms*—data-intensive scalable computing, heterogeneous computing, big data analytics, debugging, testing, refactoring, software development tools**

## I. Rise of Data-Intensive and Compute-Intensive Development

**Data Intensive Scalable Computing.** The importance of emerging data-intensive and compute-intensive applications continues to grow at an increasing rate. Cloud computing frameworks make such development widely accessible by providing readily available resources. For example, cloud services such as Amazon Web Services, Google Cloud, and Microsoft Azure make it easy to run big data applications on data-intensive scalable computing frameworks such as Google's MapReduce, Apache Spark, and Apache Hadoop. DISC frameworks enable processing massive data sets by providing distributed, parallel versions of dataflow operator implementation and allow developers to express application logic expressed in terms of user-defined functions (UDFs). In the context of this paper, we use a term, *big data analytics* or DISC applications, to refer to software that run on DISC frameworks.

**Heterogeneous Computing.** The end of Moore's law has led to a plateau in traditional single-core and software-based performance optimizations, highlighting the importance of incorporating hardware heterogeneity and specialization in software systems. To facilitate such architectures, hardware vendors support CPU+accelerator multi-chip packages (*e.g.,* Intel Xeon [1, 2], Samsung SmartSSD [3]). Concurrently, the widespread adoption of public cloud services, such as Amazon F1 [4] and Intel Devcloud [5], hold promise for integrating heterogeneous hardware resources and enabling on-demand custom hardware acceleration. Despite these resources and public's awareness, a marked disparity remains evident, as the U.S. Bureau of Labor Statistics [6] records only 74,640 hardware designers compared to 1.8 million software engineers. This significant discrepancy necessitates concerted effort on democratizing heterogeneous computing—in other words, we must reduce adoption barriers for leveraging hardware heterogeneity for typical software engineers without deep hardware expertise.

In Section III, we discuss significant challenges for making automated SE tractable in the data-intensive and compute-intensive domain, which include:

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

In Sections IV and V, we showcase examples

979-8-3503-2496-9/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-FoSE59343.2023.00006

of automated debugging, testing, and refactoring techniques that our team at UCLA has developed over the past ten years. These techniques are the results of interdisciplinary research between software engineering, big data systems, and heterogeneous computer architectures.

In Section VI, we then summarize the lessons learned from this research effort, which are summarized below:

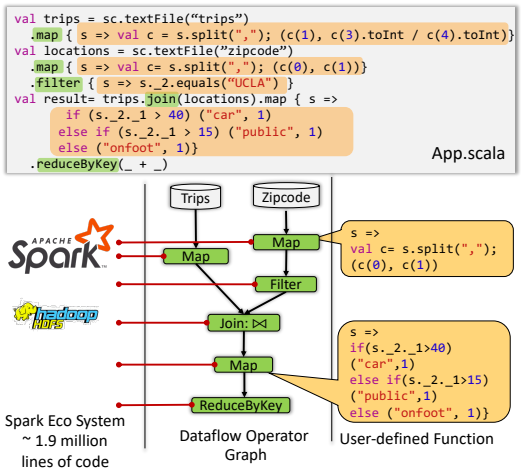
- Abstraction is necessary for speed.
- Injecting debuggability requires re-design of underlying runtimes, compilers, and systems.
- Systems-level optimizations are necessary; significant interdisciplinary engineering effort is necessary and worthwhile.
- Less is more. Winnowing out debugging results is necessary at this scale.
- Domain-specialization is necessary and we cannot wait for a large corpus to exist first.

## II. Background

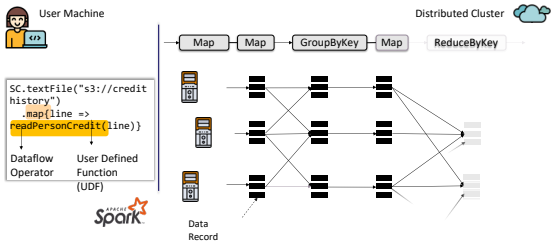
### A. Data Intensive Scalable Computing (DISC)

*Dataflow operators and user-defined functions.* DISC applications, such as the one written for Apache Spark in Figure 1a, use a combination of dataflow operators, such as `map` and `join` that take user-defined functions (UDF) as an argument. Unlike SQL queries that use a simple predicate as a user-defined function, user-defined functions in DISC applications could be arbitrarily long and complex, ranging from a few hundred lines to 100+ KLOC. Figure 1a shows an example DISC application that depends on almost 1.9 million LOC in Apache Spark. Our prior experience of designing DISC debugging tools [7, 8] indicates that most bugs appear in user-defined functions and the use of framework APIs, not the framework code itself.

Though dataflow operators have clean logical semantics, their framework implementation easily surmounts to a million lines of code, as the code is responsible for parallel executions, job scheduling, data partitioning, fault tolerance, etc, as shown in Figure 1b. Spark's runtime transforms the submitted program into smaller chunks of tasks and assign these tasks to workers to be executed on a subset of data, called partition. As a result, the start-up latency associated with invoking the *Spark framework* can take several seconds for merely setting up an execution environment. This very long latency makes it impossible to directly apply existing test generation techniques such as fuzzing to big data analytics.




(a) A data-intensive Apache Spark application relies on 1.9 millions of lines of Spark framework code, whereas the user-defined functions are hundreds of lines of code.




(b) Spark applications consist of API calls to dataflow operators such as `map` and `reduce`. Execution is distributed, complex, lazy-evaluated, and highly optimized. Spark's runtime transforms the submitted program into smaller chunks of tasks and assign these tasks to workers to be executed on a subset of data, called partition.

Fig. 1: Apache Spark: an example of data-intensive scalable computing

*Postmortem debugging based on logs.* Currently, developers do not have easy means to debug DISC applications. The use of cloud computing makes application development feel more like batch jobs and the nature of debugging is therefore postmortem, as shown in Figure 4a. Developers are notified of runtime failures or incorrect outputs after many hours of wasted computing cycles on the cloud. DISC systems such as Spark do provide execution logs of submitted jobs. However, these logs present only the physical view of big data processing, as they report the number of worker nodes, the job status at individual nodes, the overall job progress rate, the messages passed between nodes, etc. These logs do not provide the logical view of program execution e.g., system logs do not convey which intermediate outputs are produced from which inputs, nor do they indicate what inputs are causing incorrect results or delays, etc.

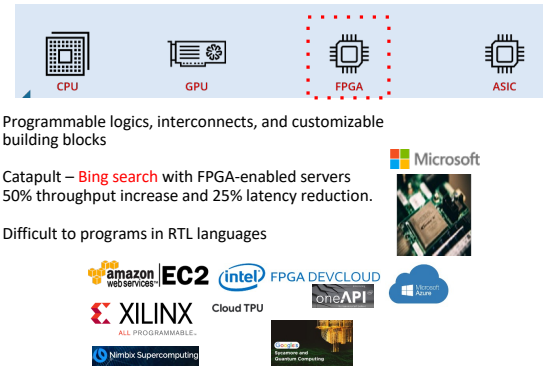



Fig. 2: A spectrum of heterogeneous hardware architectures is becoming available on cloud services

*Testing via sampling.* The standard practice for testing DISC applications today is to select a subset of inputs based on the developers' hunch with the hope that it will reveal possible defects. For example, to test big data applications, developers may use a small sample of data selected via random sampling or top $k$ sampling. Not surprisingly, such sampling is unlikely to yield adequate coverage, leading to errors in production [9]. Developers could always increase the number of samples or run the application on the entire data set stored on the cloud. However, increasing a sample size also increases testing time. More importantly, testing on the entire dataset may still be inadequate, as the application code could be continuously evolving and thus it is hard to know what should be ingested data for the evolving application a priori.

### B. Heterogenous Computing

*Cloud's shift to HW heterogeneity.* Moore's original prediction in 1965 called for a doubling in transistor density yearly. Although Moore's Law held for many decades, it began to slow sometime around 2000 and by 2018 showed a roughly 15-fold gap between Moore's prediction and current capability. Dennard scaling states that as transistor density increased, power consumption per transistor would drop so the power per square-mm of silicon would be near constant. However, Dennard scaling began to slow significantly in 2007 and faded to almost nothing by 2012. Cost efficiency, specifically, comes from leveraging the economies of scale of buying tens of thousands of mostly the same type of servers. Managing a homogeneous system is much easier from the perspective of the operating system. *So why are cloud services shifting to hardware heterogeneity?* The obvious reason is the slowdown of technology scaling, also known as the obligatory computer architect's reference to the end of Moore's Law. If applications require more compute with energy efficiency, they need to look at special purpose hardware design. Many services such as a search engine has a strict latency requirement. The tail latency target of individual micro-services is in the order of microseconds.

*Heterogeneous hardware accelerators.* We are currently entering the era of heterogeneous hardware accelerators on the cloud where cloud services are going beyond CPUs and GPUs, as shown in Figure 2. Amazon F1 at Amazon Web Services and Intel One API are such examples. For example, Microsoft's Catapult and Brainwave are built on reconfigurable fabrics built on FPGA. Expensive hardware such as supercomputers and quantum computers are also being offered as a public cloud service. This is attractive for developers and users—Pay as you go—as there is no need to buy and maintain expensive hardware.

The main advantage of CPUs is that it is very easy to program them and supports any programming framework. GPUs are specialized processing units that were mainly designed to process images and videos. GPUs are programmed in languages like CUDA and OpenCL and therefore provide limited flexibility compared to CPUs. FPGAs are Field Programmable Gate Arrays. In the past, FPGAs used to be a configurable chip that was mainly used to implement glue logic and custom functions. However, currently FPGAs have been emerged as a very powerful processing units that can be reprogrammed to meet applications' requirements with lower cost and lower power consumption. ASICs are application-specific integrated circuits. For ASIC, its design cycle is long at around 6 months to several years; thus, it is impractical to spin up ASIC for each algorithm or every evolving application.

*High level synthesis for FPGA.* We discuss FPGA in more detail, as it is re-programmable hardware. FPGAs are high-performance hardware devices that can be customized to accelerate compute-intensive software [10, 11, 12] across a wide variety of domains, including data science and machine learning [13, 14, 15]. With FPGAs, developers can create *heterogeneous applications* that consist of both *host* code and *kernel* code, where compute-intensive kernels can be offloaded from CPU to FPGA accelerators. Although FPGAs provide substantial benefits and are commercially available to a broad user base, they are associated with a high learning bar-

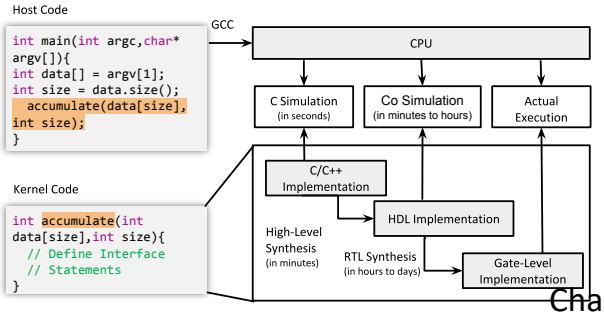
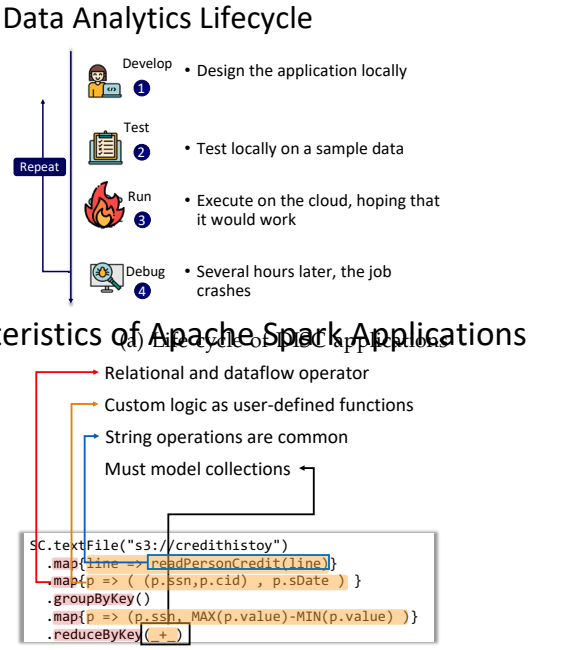



Fig. 3: High level synthesis development work flow: simulation and synthesis range 5 minutes and 3.5 hours on average for the popular HLS Rosetta benchmark [19].




(b) Characteristics of DISC applications

Fig. 4: Data-intensive scalable application development

rier [16]. Programming an FPGA is a difficult task; hence, it is limited to a small subset of programmers with deep knowledge of micro-architecture details.

To address this issue, there has been work on high-level synthesis (HLS) compilers for FPGAs [17]. HLS tools take a kernel written in C/C++ as input and automatically generates an FPGA accelerator. For example, Intel/Altera SDK for OpenCL and Xilinx Vivado HLS [17, 18] automatically generate Register-Transfer Level (RTL) descriptions in the form of a bitstream from code written in C/C++.

During HLS, the front-end process generates an RTL description, which is then sent to its backend to *schedule* each operation from the kernel code to specific clock cycle time slots. Next, it allocates the number and type of hardware unit *resources* used for implementing functionality, like loop-up-tables (LUTs), flipflops (FFs), Block RAM (BRAMs), digital signal processing units (DSPs), etc. Finally, the *binding* stage maps all operations to the allocated hardware units. As illustrated in Figure 3, given a heterogeneous application with host code and kernel code, the build and execution process involve: (1) **C simulation** which runs the program as a C program; (2) **co-simulation** which runs the host code on CPU but maps the kernel calls to the simulator calls with the generated RTL by HLS; and (3) **hardware execution** which runs the host code on CPU and kernel code on hardware with generated bitstream by RTL synthesis. This process can take several minutes for simulation and several hours for synthesis, depending on kernel logic complexity.

## III. Challenges

### A. Testing

The long latency of data-intensive and compute-intensive applications prohibits the applicability of fuzzing. Fuzzing normally requires thousands of program invocations in a second; however, the minimum execution time of a big data application running on Apache Spark is 10 seconds just for the Spark context set up. The simulation time of an FPGA-based heterogeneous application is over at least 2 minutes. As a result, AFL[20]-like naïve fuzzing would spend 98% of the time setting up a test environment.

Conventional guidance metrics and low-level mutation operators are unlikely to scale for the data-intensive and compute-intensive domain. A significant chunk of big data analytics code comes from the DISC framework implementation (*e.g.*, 1.9MLOC for Apache Spark). Therefore, not being able to distinguish framework code vs. application code during coverage monitoring could easily target more testing of the underlying framework, not the application logic. Furthermore, to analyze DISC applications, we must reason about the semantics of relational and dataflow operators such as `map`, `groupByKey`, `reduceByKey`, shown in pink in Figure 4b. Custom application logic is often expressed as user-defined functions (UDF) shown in orange. String operations are common as DISC applications often process unstructured, semi-structured, or structured data sets. Due to frequent use of aggregators, we must also model collections. Therefore, symbolic testing techniques must account for the use of dataflow operators, the semantics of UDFs, and symbolic modeling of

strings and collections.

Our investigation has found that most bugs appear in the user application, not the framework implementation. Similarly, in the domain of heterogeneous computing, traditional code coverage as a fuzzing guidance metric cannot account for hardware accelerator synthesis assumptions that are often the root cause of divergent behavior between CPU and custom accelerators. Due to lack of guidance signals at the hardware level, testing applications that run on real heterogeneous architectures is extremely challenging as kernels are black boxes, providing no information about the kernels' internal execution to diagnose issues such as silent hangs or unexpected results.

Random bit or byte-level mutations can hardly generate meaningful data capable of revealing real-world bugs in these domains as well. For example, flipping a random bit in the input is unlikely to generate the expected tabular or matrix input format, resulting in unnecessary invocation or early termination of the target program. Additionally, traditional iterative fuzz testing techniques often mutate a small part of a seed input to generate new inputs. While this approach works well for many CPU programs, it is extremely ineffective for heterogeneous applications. Inputs of heterogeneous applications are often large matrices and tensors, leading to significant data access and transfer overheads —the host, which mutates matrices, must send newly mutated matrices (e.g., with only a few elements modified) to the device. For a 100k×100k matrix, a single process of offloading the newly generated matrix from CPU to the device would take 2 minutes, prohibiting fast fuzzing on heterogeneous architectures.

### *B.* **Debugging**

Currently, debugging big data analytics is therefore an ad-hoc, time-consuming process. When a job fails or they get results that end up being suspicious, data scientists must identify the source of the error, often by digging through post-mortem logs. In such cases, the programmer may want to pinpoint the root cause of errors by investigating a subset of corresponding input records. Due to the scale of large data, manually sift through a terabyte size data set is clearly time-consuming and infeasible.

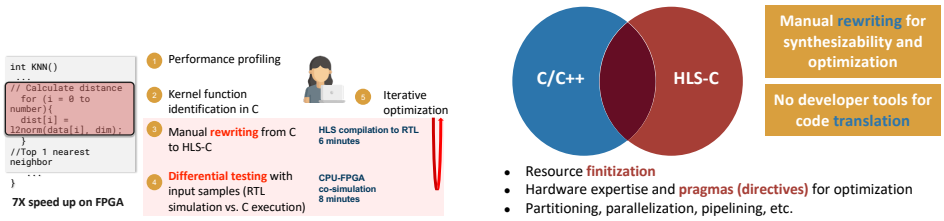
Similarly, debugging heterogeneous applications is similarly challenging due to the lack of observability. Traditional hardware simulation waveforms rarely help developers reflect faults at the software or data level. Developers in heterogeneous computing domain often have questions about *"what caused my program to produce unexpected results and to slow down?"* In practice, most HLS designers are accustomed to using simulation waveforms to debug their accelerators and to estimate execution time at a cycle level. However, it is extremely difficult to locate faults in developers' source code based on hardware execution or waveforms from hardware simulation. There are two primary causes: (1) The non-monotonic code increase during the HLS compilation process reduces source code traceability from C/C++ to hardware bitstreams, and (2) developers lack deep understanding of heterogeneous system stacks and numerous customizable hardware configurations, which have a confounding effect on ensuring correctness.

### *C.* **Refactoring**

While HLS compilers take kernel code in C dialects, a developer must perform a substantial amount of manual refactoring to make it *synthesizable* (*i.e.,* hardware-compatible) and *efficient*, as shown in Figure 5a. For example, all recursions produce compilation errors in HLS and must be converted into iterations using a stack with a finite estimated size (*i.e.,* resource finitization on hardware). Additionally, developers must insert synthesis directives and pragmas in their source code manually to achieve good performance. Technically HLS-C is a dialect of C and is not the same as regular C/C++, as significant manual rewriting is required for synthesizability and optimization. This requires having inter-disciplinary expert knowledge and knowing obscure platform-dependent details. For example, developers need to know micro-architecture level details to decide on how to parallelize and pipeline computation, how to partition data arrays to map to on-chip memory blocks, etc. Most software programmers do not know how to perform these hardware-specific optimizations.

## IV. Example Techniques for Data Intensive Scalable Computing

For the past ten years, our team at UCLA have worked on extending and adapting software debugging and testing techniques to the domain of big data analytics written in Apache Spark [7, 9, 21, 22, 23, 24, 25, 26, 27]. From this experience, we have learned that designing interactive debug primitives for a dataflow based big data system requires deep understanding of an internal execution model, job scheduling, and materialization;




(a) Developer workflow with high level synthesis

(b) HLS tools are not easy to use.

Fig. 5: High level synthesis (HLS)

providing traceability requires re-engineering an underlying data-parallel runtime framework; and abstraction is a powerful force in simplifying code paths and reducing the latency of test execution.

### A. BigDebug: Interactive Debug Primitives for Big Data Analytics

We have had tools such as GDB for a long time. So why is hard to build an interactive debugger for Apache Spark? Naïve implementation of breakpoints would not work, because pausing the entire computation in the data-parallel pipeline reduces throughput and it is clearly infeasible for a user to inspect billion of records through a regular watchpoint. BigDebug [7] does not pause program execution but instead simulates a breakpoint through on-demand state regeneration from the latest checkpoint and delivers program states in a guarded, stream processing fashion. By effectively tapping into internal checkpointing and job scheduling mechanisms, we were able to implement interactive debugging and repair capability in Apache Spark efficiently, while adding at most 34% overhead [7].

### B. Titian: Data Provenance for Apache Spark

Data provenance is a long studied problem in databases. Given an output of query, data provenance identifies specific inputs contributing to the query results. The idea is similar to dynamic taint propagation. For big data analytics with terabyte data, scalability poses a new challenge. To provide record level data provenance, we re-engineered Apache Spark's runtime by storing lineage tables (the input and output tag mappings) at a stage granularity in a distributed manner and building a distributed optimized join for backward tracing, which is order of magnitude faster than alternatives [22].

### C. BigSift: Automated Debugging of Big Data Analytics

BigSift takes a program and a test function as inputs, and automatically finds a minimum subset of inputs producing test failures. BigSift combines two mature ideas—data provenance in DB and delta debugging in SE—and implements several optimizations: (1) test predicate pushdown, (2) prioritizing backward traces, and (3) bitmap based memorization, which enabled us to build an automated debugging solution that is 66X faster than delta debugging and takes 62% less time than the original job's run [21].

### D. BigTest: White-Box Testing of Big Data Analytics

Currently, developers sample data (e.g., random sampling, top n sampling, and top k% sampling) to test data analytics, which leads to low code coverage. Another option is to use traditional test generation such as symbolic execution but such technique would not scale for Apache Spark (about 700 KLOC).

To automatically generate tests for a Spark application, BigTest abstracts dataflow operators, in terms of clean first order logic [9]. For example, `join` could be defined as three equivalence classes where a key is only present in the left table, the right table, and neither. Then for a user defined application code, BigTest performs symbolic execution and combines it together with dataflow logical specifications. These combined constraints are called as JDU path constraints (joint dataflow and UDF constraints) as shown in Figure 6 and they are solved using SMT to create concrete inputs. Only 30 or so records are required to achieve the same code coverage as the entire data, implying that testing on the entire data is not necessary. By automatically generating data with BigTest, we can reduce the required test data by $10^8$, achieving almost 200X speed up [9].

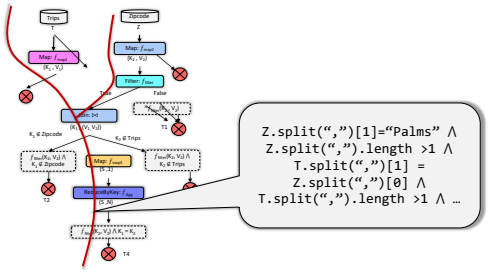



Fig. 6: Joint Dataflow and UDF Path in a DISC application

### E. PerfDebug: Performance Debugging of Computation Skews

Performance is a key factor for big data applications, and much research has been devoted to optimizing these applications. When an application shows signs of poor performance through an increase in general CPU time, garbage collection (GC) time, or serialization time, the first question a user may ask is *“what caused my program to slow down?”* While prior work can diagnose and correct *data skew*, the problem of *computation skew*—abnormally high computation costs for a small subset of input data—has been largely overlooked.

PERFDEBUG [24] is a post-mortem performance debugging tool that enables a user to debug applications that exhibit computation skew. It automatically finds input records responsible for such abnormalities in a big data application by reasoning about deviations in performance metrics such as job execution time, garbage collection time, and serialization time. The key enabler behind PERFDEBUG is a data provenance-based technique that computes and propagates *record-level computation latency* to keep track of *abnormally expensive records* throughout the pipeline. The input records that have the largest latency contributions are then presented to the user for bug fixing.

### F. FlowDebug: Influence-based Provenance with Taint Propagation

FLOWDEBUG [25] further improves data provenance using two insights. First, precisely tracks control and data flow within user-defined functions to propagate taints at a fine-grained level by inserting custom data abstractions through automated source to source transformation. Second, it introduces a novel notion of influence-based provenance for many-to-one dependencies to prioritize which input records are more *responsible* than others by analyzing the semantics of a user-defined function used for aggregation. FLOWDEBUG significantly improves the precision of debugging results by up to 99.9 percentage points and avoids repetitive reruns required for post-mortem analysis by a factor of 33 while incurring an instrumentation overhead of 0.4X - 6.1X on vanilla Spark.

### G. OptDebug: Operation Provenance with Spectra-based Fault Localization

Data provenance is concerned with fault isolation only in the data-space, as opposed to fault isolation in the code-space—*how can we precisely localize operations or APIs in code responsible for a given suspicious or incorrect result?*

OPTDEBUG [26] identifies fault-inducing operations (i.e., APIs) in code in a dataflow application using three insights. First, debugging is easier with a small-scale input than a large-scale input. So, it uses data provenance to simplify the original input records to a smaller set leading to test failures and test successes. Second, keeping track of *operation provenance* is crucial for debugging. Thus, it leverages automated taint analysis to propagate the lineage of operations downstream with individual records. Lastly, each operation may contribute to test failures to a different degree. Thus, OPTDEBUG ranks each operation’s *spectra*—the relative participation frequency in failing vs. passing tests. In our experiments, OPTDEBUG achieves 100% recall and 86% precision in terms of detecting faulty operations and reduces the debugging time by 17× compared to a naïve approach.

### H. BigFuzz: Fuzz Testing using Framework Abstraction

BIGFUZZ [27] is a coverage-guided fuzz testing tool for data-intensive applications. It focuses on exercising application logic instead of increasing framework code coverage by abstracting the framework using specifications. The key insight behind BIGFUZZ is that *fuzz testing of data-intensive applications can be made tractable by abstracting framework code and by analyzing application logic in tandem*. The key idea is to perform source-to-source transformation of a data-intensive application and generate a semantically equivalent yet framework-independent program that is more amenable to fuzzing. Figure 7 illustrates BIGFUZZ’s approach and three key components to reduce fuzzing latency, to construct error-type guided mutations, and to design a new guidance metric.

BIGFUZZ performs automated source to source transformation to construct an equivalent DISC application suitable for fast test generation. It introduces schema-aware data mutation operators based

on an in-depth study of dataflow application error types. Through an extensive evaluation, we have shown that BIGFUZZ speeds up the fuzzing time by 78 to 1477X compared to random fuzzing, improves application code coverage by 20% to 271%, and achieves 33% to 157% improvement in detecting 81% more application bugs.

## V. EXAMPLE TECHNIQUES FOR HETEROGENEOUS COMPUTING

Specialized hardware accelerators like GPUs and FPGAs become a prominent part of the current computing landscape. However, developing heterogeneous applications is limited to a small subset of programmers with specialized hardware knowledge. To democratize heterogeneous computing, for the past four years, our team at UCLA has worked on automated refactoring, testing, and debugging tools for heterogeneous application development [28, 29, 30, 31, 32]. From this experience, we have learned that deep understanding of underlying hardware platforms' design and optimization assumptions can improve effectiveness of automated SE tools.

### A. HeteroRefactor: Refactoring for Heterogeneous Computing for FPGA

High-level synthesis (HLS) tools made significant progress in raising the level of programming abstraction from hardware programming languages to C/C++, but they usually cannot compile and generate accelerators for kernel programs with pointers, memory management, and recursion, and require manual refactoring to make them HLS-compatible. Besides, experts also need to provide heavily handcrafted optimizations to improve resource efficiency, which affects the maximum operating frequency, parallelization, and power efficiency. For example, as shown in Figure 9a, HLS requires specifying bit-width for each data type. Instead of using default 32 bits on CPU, HLS uses arbitrary bitwidths. For example, one can use 7 bit integers for her program. Similarly, a developer needs to tune the precision of floating point numbers. Instead of using default 8bits exponent and 23 bit mantissa for a 32 bit floating points (IEEE FP standard), HLS developers have the flexibility of tuning precision for memory saving, as shown in Figure 9b. Recursion, pointer, dynamic memory allocation such as `malloc` and `free` in standard C, are not available in HLS, so developers must rewrite their application using a finite size array, as shown in Figure 9c. In fact, such C to HLS-C refactoring is extremely time-consuming and without manual optimizations, performance boost is not automatic. A recent article [33] reports that, for a 7 line convolution neural network example C code, an existing commercial HLS tool generates an FPGA-based accelerator 108× slower than a single-core CPU. Then after proper re-structuring of the input C code (to tile the computation, for example) and inserting 28 pragmas, the final FPGA accelerator is 89× faster than a single-core CPU.

HETEROREFACTOR [28] performs automated refactoring for HLS-based heterogeneous applications by leveraging FPGA-specific dynamic invariant detection. First, HETEROREFACTOR monitors the required bitwidth of integer and floating-point variables, and the size of recursive data structures and stacks. Second, using this knowledge of dynamic invariants, it refactors the kernel to make traditionally HLS-incompatible programs synthesizable and optimize the accelerator's resource usage and frequency further. Third, to guarantee correctness, it selectively offloads the computation from CPU to FPGA, only if an input falls within the dynamic invariant. On average, for a recursive program of size 175 LOC, an expert FPGA programmer would need to write 185 more LOC to implement an HLS compatible version, while HETEROREFACTOR automates such transformation. Our results on Xilinx FPGA show that HETEROREFACTOR minimizes BRAM by 83% and increases frequency by 42% for recursive programs; reduces BRAM by 41% through integer bitwidth reduction; and reduces DSP by 50% through floating-point precision tuning.

### B. HeteroGen: Transpiling C to Heterogeneous HLS Code

HeteroGen [29] is an automated repair tool that takes as input a regular C/C++ program and produces its HLS-C counterpart *without involving any human developer in the loop.* On one hand, HETEROGEN is a *transpiler* that performs behavior-preserving source-to-source translation from C/C++ to HSL-C by automatically resolving compatibility issues; on the other hand, it is an *optimizer* that checks whether the updated code has superior performance than the original version. Although HETEROGEN does not guarantee to generate *optimal* code, it represents a best-effort approach to produce the highest level of HLS compatibility and efficiency improvement within a time budget. Figure 10 illustrates HETEROGEN's approach.

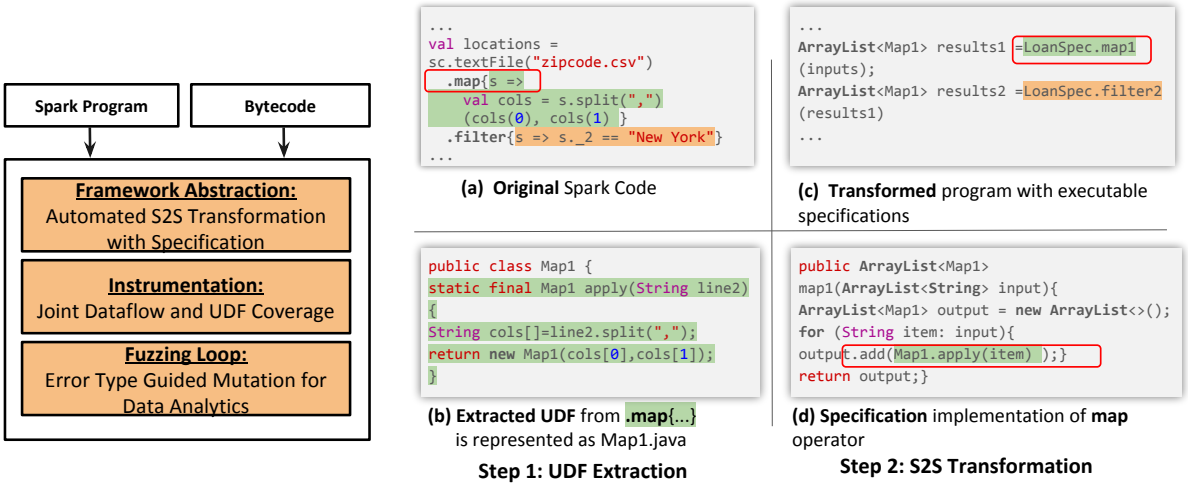
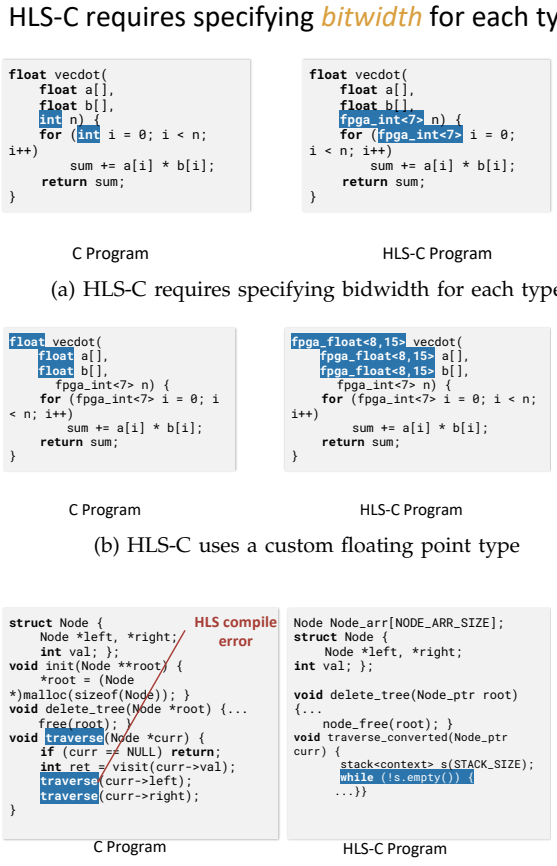



Fig. 8: BigFuzz: Improving fuzzing using executable specifications of dataflow implementation




(a) HLS-C requires specifying bidwidth for each type

(b) HLS-C uses a custom floating point type

(c) HLS-C requires finitizing resources

Fig. 9: HLS rewriting examples

To repair compatibility errors, there is a huge search space (of possible program edits). To tackle this challenge, HETEROGEN leverages *common HLS repair patterns*. With a study of more than 1,000 posts from Xilinx's HLS Q&A forum, we summarize six common repair patterns, regarding dynamic data structures, unsupported data types, dataflow optimization, loop parallelization, struct and union, and top functions. These edit patterns are encoded as parameterized *repair templates* at the level of abstract syntax trees. To overcome the challenge of long HLS compilation time, HETEROGEN leverages a lightweight LLVM-based checker to validate repairs. Our key insight here is that *if a repair does not conform to HLS coding styles, it does not need to be compiled*.

Its evaluation shows that HETEROGEN could repair all of HLS compatibility errors, with an average of 2,437 tests generated per application, achieving branch coverage of 97%. It automated 9 to 438 lines of edits to produce an HLS version, which is, on average, 1.63× faster than the original C version.

### C. HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence

HETEROFUZZ [30] targets fuzzing of heterogeneous applications to detect platform-dependent divergence. The key essence of HETEROFUZZ is to reduce the long latency of repetitively invoking a hardware simulator on a heterogeneous application. First, in addition to monitoring code coverage as a fuzzing guidance mechanism, it analyzes synthesis pragmas in kernel code and monitors

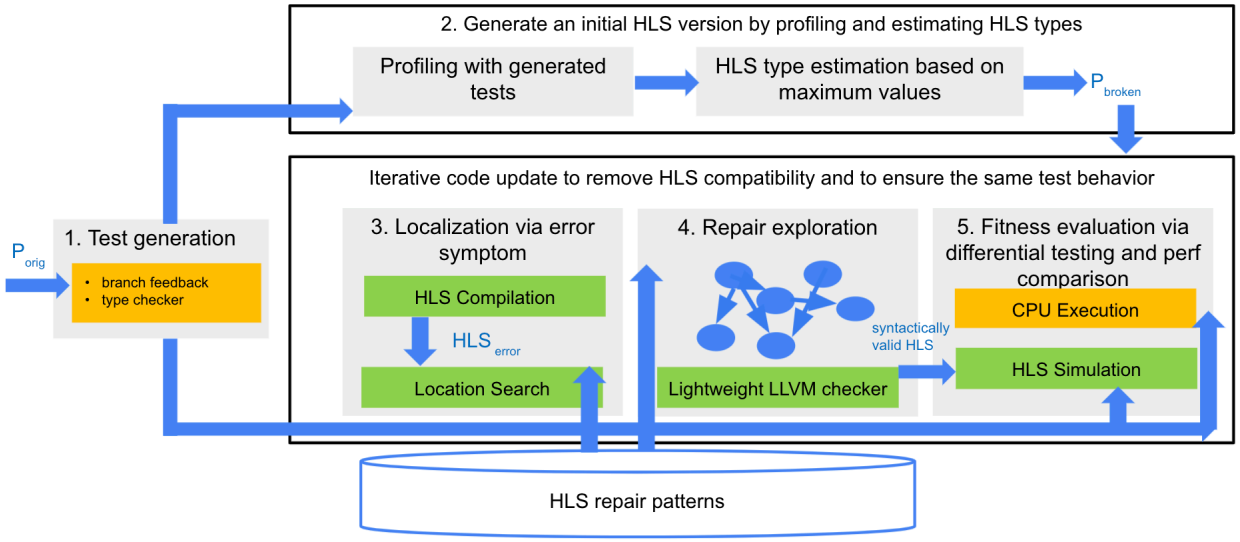



Fig. 10: HETEROGEN takes as input an original kernel program ($P_{orig}$). It auto-generates test inputs for $P_{orig}$, and an initial version $P_{broken}$ with estimated HLS types. Next, it finds repair locations based on an HLS error symptom, explores the space of applicable repairs based on fix patterns, and evaluates behavior preservation via differential testing.

accelerator-relevant value spectra. Second, it uses dynamic probabilistic mutations to increase the chance of hitting divergent behavior under different platforms. Third, it memorizes the boundaries of seen kernel inputs and skip HLS simulator invocation if it can expose only redundant divergent behavior. On seven real-world heterogeneous applications with FPGA kernels, HETEROFUZZ is 754× faster in exposing the same set of distinct divergence symptoms than naive fuzzing. Probabilistic mutations contribute to 17.5× speed up than the one without. Selective invocation of HLS simulation contributes to 8.8× speed up than the one without.

### D. HFuzz: Leveraging Hardware Probes and Optimizations for Accelerating Fuzzing

Testing applications on *real* heterogeneous architectures is extremely challenging, as kernels are black boxes, providing no information about their internal hardware execution to diagnose issues such as silent hangs or unexpected results. HFUZZ [31] leverages the *capability of heterogeneous hardware* for testing *heterogeneous applications*. HFUZZ increases both the observability of hardware kernels and testing efficiency through a three-pronged approach. It inserts device-side in-kernel hardware probes in addition to host-side software monitors. Second, it performs rapid input space exploration by offloading compute-intensive input mutations to hardware kernels. Third, it parallelizes fuzzing and enables fast on-chip memory access, by utilizing four FPGA-level optimizations including loop unrolling, shannonization, data preloading, and dynamic kernel sharing.

On Intel OneAPI subject programs, HFUZZ speeds up fuzz testing by 4.7× with HW-accelerated input space exploration. By incorporating HW probes in tandem with SW monitors, HFUZZ finds more defects and reveals more unique, unexpected behavior symptoms that could not be found by SW-based monitoring alone.

### E. QDiff: Differential Testing of Quantum Software Stacks

Quantum computing continues to grow in popularity as an alternative heterogeneous hardware, as quantum supremacy for certain classes of algorithms has now been illustrated. QDIFF [32] is an automated testing framework for quantum compilers and simulators. There are three main challenges for testing quantum software stacks including (a) generating semantically equivalent programs for testing compilers, (b) exposing and examining bugs in quantum simulators and hardware, and (c) interpreting measurements from the results of running quantum programs given their probabilistic nature. To address these issues, QDIFF generates logically equivalent quantum programs using a set of equivalent gate transformation rules, selects subsets of said programs to run on hardware, and determines

how many measurements are needed for a reliable comparison between equivalent programs to detect potential issues.

QDIFF was evaluated with three widely-used open source QSSes: Qiskit from IBM, Cirq from Google, and Pyquil from Rigetti. QDIFF found several critical bugs revealing potential instabilities in these platforms.

## VI. LESSONS LEARNED

We discuss the lessons that we learned by adapting automated debugging, testing, and refactoring methods to data intensive scalable computing and heterogeneous computing. In particular, we share *broken assumptions*, a set of assumptions that typically hold when designing an automated software engineering tool that no longer hold in the data-intensive and compute-intensive domain.

Lesson 1: Abstraction is useful for increasing speed.

Based on our experience of designing BIGTEST [34] and BIGFUZZ [27], we learned that abstraction is necessary not only for taming complexity but also for increasing speed. The key idea of BIGFUZZ was to leverage abstraction to increase speed by applying source-level API rewriting for dataflow and relational APIs. The key idea was of BIGTEST was to leverage abstraction to simplify path constraints by combining UDF symbolic execution with dataflow and relational operator specifications.

When it comes to test input generation, symbolic execution is generally considered to be not suitable for real-world size applications due to poor scalability. On the other hand, fuzzing is often shown to be effective in practice. However, the effectiveness of fuzz testing is built on the implicit assumption that a subject program under test must run super fast (say a few milliseconds per invocation) to allow a huge number of repetitive invocations in a short time span. In the context of data-intensive and compute-intensive domain, we observed none of these assumptions about test generation are true, because the time to initialize DISC framework adds significant latency (e.g., 15 seconds to initialize Spark context), preventing the applicability of naive fuzzing. Despite the size of a DISC application code, on the other hand, symbolic execution is feasible, when we abstract dataflow operators with logical specifications, since the semantics of dataflow operators such as `map` and `reduceByKey` are stable and thus their specifications do not change.

*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.

When we were initially designing an interactive debugger [7] and data provenance support for Apache Spark [35], we encountered common misconceptions: injecting traceability into distributed system runtimes would add intolerable overhead and replay debugging is too slow to be practical.

However, in the BIGDEBUG project [7], we demonstrated that adding data provenance adds about 25% overhead and this tradeoff is worthwhile, since developers could gain visibility into opaque computation in return. We also demonstrated that if we replay the computation from the latest checkpoint such as the latest *materialization point at the shuffle boundary* that already exist in Spark's runtime system, we do not need to add much extra overhead. Similarly, in the PERFDEBUG project [24], we demonstrated that fine-grained latency tracking is feasible by rewriting Spark's runtime to enable performance lineage tracking.

*Broken assumptions: Injecting traceability and debuggability adds intolerable overhead.*

Lesson 3: Systems-level innovations and optimizations are absolutely necessary

The idea of data provenance in Titian [35] is essentially the same as dynamic taint tracking in software engineering. The key idea behind BigSift [21] is essentially delta debugging, a 20+ years old, well-known technique. However, implementing these simple ideas at the scale of terabyte data requires significant innovations in terms of systems level optimization. In TITIAN [35], we had to implement distributed, optimized, backward recursive join via partition-id tracking to make data provenance efficient and scalable. In BIGSIFT [8], we had to also implement systems-level optimizations for memoizing similar executions and pushing oracle evaluation to earlier computation stages when applying delta debugging to DISC applications. BIGSIFT is 66× faster than delta debugging and takes 62% less time to debug than the original job's run due to these optimizations.

Lesson 4: Less is more

In software engineering and program analysis, there is natural gravity towards soundness and completeness. Completeness in debugging is often

interpreted as finding all causes (or all inputs) that affect the buggy outcome. However, such completeness may be unnecessary or even be counterproductive at the scale of huge data.

In FLOWDEBUG [25], when implementing taint analysis with an influence function, we learned that the use of aggregation operators such as `reduceByKey` to compute `sum` or `average` inevitably makes *all* inputs contributing to the aggregated outcome. Similarly, in OPTDEBUG [26], when implementing spectra-based fault localization for DISC applications, we learned that winnowing out a large portion of failure-inducing inputs via operation-level tainting is necessary not to overwhelm developer's attention span.

*Broken assumptions: conservatively identifying all inputs that affect a given outcome is important.*

Lesson 5: Domain-specialization is necessary, yet no large corpus exists for a new domain.

In HETEROREFACTOR [28], when building an automated refactoring for heterogeneous applications, we had to encode domain specific knowledge when optimizing bitwidths and FP precision for the best accuracy and performance tradeoffs. In HETEROFUZZ [30], when designing a test input generation technique for heterogeneous applications, we had to account for hardware behavior by designing HW-specific monitoring criteria (i.e., accelerator spectra) to be used as fuzzing guidance signals. In HETEROGEN [29], we had to encode common edit recipes to make automated HLS compatibility error repair feasible.

The challenge that we must face is that such domain-specialization cannot be easily delegated to the power of machine learning or large language models, since a large corpus does not exist for a yet-to-be-democratized domain. For example, the corpus statistics for InCoder [36] in Figure 11 shows that most training data comes from popular programming languages such as Python and JavaScript. On the other hand, Scala or Rust are marginalized in the representation. Languages for heterogeneous application development such as HLS-C hardly appear in the training corpus due to lack of available data.

*Broken assumptions: Many examples exist to allow machine learning or mining software repositories methods to infer domain-specific knowledge.*

## VII. FUTURE OPPORTUNITIES

**From targeting heterogeneity to leveraging heterogeneity.** Slow program execution is one challenge

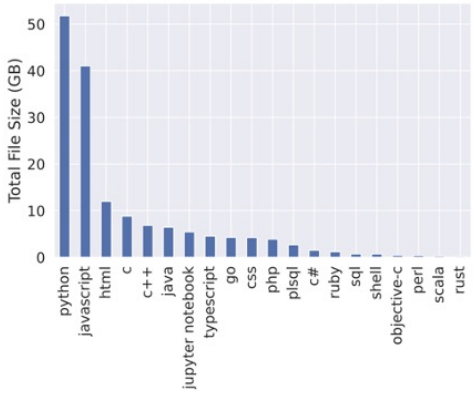



Fig. 11: Incoder Corpus Statistics [36]

in applying automated test generation in both data intensive scalable computing and heterogeneous computing domains. However, there is an opportunity to leverage hardware parallelism directly for test generation purposes. Test generation is a typically iterative process, where repetitive invocation of a subject program is required, and input mutations are applied repetitively. Therefore, there is an opportunity to leverage hardware parallelism to expedite fuzz testing. Further, there is also an opportunity to extract high fidelity feedback signals at the hardware level by injecting in-kernel probes to hardware devices.

**Targeting a system of systems not a standalone system.** Increasingly, software system stacks for data intensive scalable computing and heterogeneous computing are becoming increasingly built on layers of extensions. For example, the CIRCT project [37] shown in Figure 12 are built on multiple layers of compiler extensions fueled by a new LLVM MLIR [38] project that supports hybrid IR to enable different IR requirements in a unified infrastructure. Debugging and testing such multi-layer compilers and systems is challenging, because each layer of extension is also rapidly evolving in terms of IR representations and operator semantics. This trend thus necessitates the needs of expanding the scope of debugging from a single system to an ecosystem of systems; in other words, one must isolate the root cause for a cascading effect of a bug—*an infection chain* across multiple layers or compiler extensions, runtimes, and systems.

**Active incorporation of human feedback, while reducing inspection effort.** As we discussed in Section V, domain specialization of SE techniques is often necessary, yet a large corpus does not exist yet for such a yet-to-be-democratized domain such

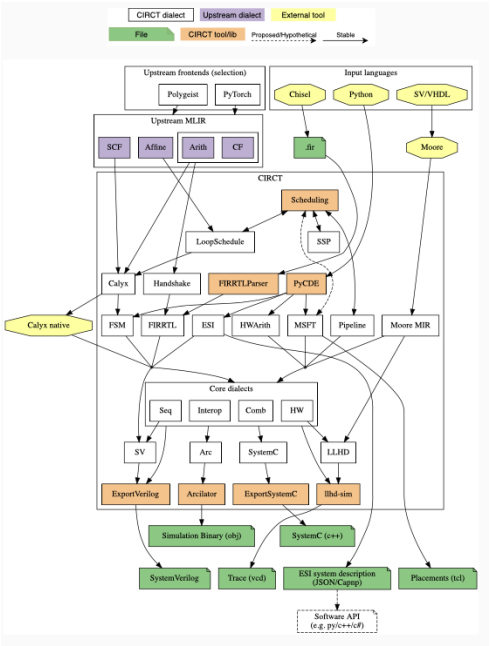



Fig. 12: CIRCT MLIR Dialects

as heterogeneous computing with FPGA. A similar case holds true for the data intensive computing domain, where sharing or public release of a DISC application code is discouraged due to privacy concerns about the input data that the application ingests.

To make fuzz testing effective in heterogeneous computing, we need custom mutation operators, domain-specific search strategies, custom feedback guidance signals, etc. We anticipate that such domain-specific knowledge should be inferred from example test cases, example code snippets, or example type declarations associated with pragmas. In order to make automated repair effective in heterogeneous computing, one needs custom fitness functions or custom repair (/fix) patterns. Active learning from example patches and human-in-the-loop design space exploration may provide advances in this area.

## Acknowledgment

This paper is an accompanying paper for an invited talk at ICSE 2023 Future of Software Engineering, titled as "Software Engineering for Big Data and HW Heterogeneity." I would like to thank my students, postdocs, and collaborators who taught me and supported me in this new direction of software development tools for data intensive scalable computing and heterogeneous computing.

This work is supported by the National Science Foundation grants 1764077, 1956322, 2106838, 2106404 and ONR grant N00014-18-1-2037. It is also supported in part by grant from Amazon Science Hub.

## References

[1] Prabhat Gupta. Xeon+fpga platform for the data center. https://www.archive.ece.cmu.edu/~calcm/carl/lib/\exe/fetch.php?media=carl15-gupta.pdf, 2019.

[2] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2018.

[3] Samsung SmartSSD, 2023.

[4] Amazon.com. Amazon ec2 f1 instances: Run custom fpgas in the aws cloud. https://aws.amazon.com/ec2/instance-types/f1, 2019.

[5] Intel. Devcloud for one api. https://devcloud.intel.com/oneapi/, 2023.

[6] Bureau of Labor Statistics, 2023.

[7] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[8] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, pages 520–534, 2017.

[9] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. White-box testing of big data analytics with complex user-defined functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 290–301, New York, NY, USA, 2019. ACM.

[10] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov,

Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[11] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. page 13–24, 2014.

[12] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–135. IEEE, 2019.

[13] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160. ACM, 2014.

[14] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. Smem++: A pipelined and time-multiplexed smem seeding accelerator for genome sequencing. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 210–2104, Aug 2018.

[15] Zhe Chen, Hugh T Blair, and Jason Cong. Lanmc: Lstm-assisted non-rigid motion correction on fpga for calcium image stabilization. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 104–109, 2019.

[16] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC*, pages 1102–1105. IEEE, 2011.

[17] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[18] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[19] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[20] American fuzz loop. http://lcamtuf.coredump.cx/afl/, 2020.

[21] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 520–534, New York, NY, USA, 2017. ACM.

[22] Matteo Interlandi, Ari Ekmekji, Kshitij Shah, Muhammad Ali Gulzar, Sai Deep Tetali, Miryung Kim, Todd Millstein, and Tyson Condie. Adding data provenance support to apache spark. *The VLDB Journal*, Aug 2017.

[23] Matteo Interlandi, Sai Deep Tetali, Muhammad Ali Gulzar, Joseph Noor, Tyson Condie, Miryung Kim, and Todd D. Millstein. Optimizing interactive development of data-intensive applications. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 510–522, 2016.

[24] Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. Perfdebug: Performance debugging of computation skew in dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 465–476, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Jason Teoh, Muhammad Ali Gulzar, and Miryung Kim. Influence-based provenance for dataflow applications with taint propagation. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 372–386, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Muhammad Ali Gulzar and Miryung Kim. Optdebug: Fault-inducing operation isolation

for dataflow applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 359–372, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[28] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. Heterorefactor: Refactoring for heterogeneous computing with fpga. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 493–505, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. Heterogen: Transpiling c to heterogeneous hls code with automated test generation and program repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1017–1029, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Qian Zhang, Jiyuan Wang, and Miryung Kim. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 242–254, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Jiyuan Wang, Qian Zhang, Hongbo Rong, Harry Xu, and Miryung Kim. Leveraging hardware probes and optimizations for accelerating fuzz testing of heterogeneous applications. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*, pages 1–12, 2023.

[32] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. Qdiff: Differential testing of quantum software stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 692–704, 2021.

[33] Jason Cong. Toward democratized ic design and customized computing. https://semiengineering.com/toward-democratized-ic-design-and-customized-computing/, 2022.

[34] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. White-box testing of big data analytics with complex user-defined functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 290–301, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[36] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2023.

[37] CIRCT Contributors. Circt: Circuit ir compilers and tools, 2023.

[38] Multi-level ir compiler framework. https://mlir.llvm.org, 2023.