HeteroGen: Transpiling C to Heterogeneous HLS Code with Automated Test Generation and Program Repair

Qian Zhang
zhangqian@cs.ucla.edu
University of California, Los Angeles
USA

Jiyuan Wang
wangjiyuan@cs.ucla.edu
University of California, Los Angeles
USA

Guoqing Harry Xu
harryxu@cs.ucla.edu
University of California, Los Angeles
USA

Miryung Kim
miryung@cs.ucla.edu
University of California, Los Angeles
USA

ABSTRACT

Despite the trend of incorporating heterogeneity and specialization in hardware, the development of heterogeneous applications is limited to a handful of engineers with deep hardware expertise. We propose HeteroGen that takes C/C++ code as input and automatically generates an HLS version with test behavior preservation and better performance. Key to the success of HeteroGen is adapting the idea of search-based program repair to the heterogeneous computing domain, while addressing two technical challenges. First, the turn-around time of HLS compilation and simulation is much longer than the usual C/C++ compilation and execution time; therefore, HeteroGen applies *pattern-oriented program edits* guided by common fix patterns and their dependences. Second, behavior and performance checking requires testing, but test cases are often unavailable. Thus, HeteroGen auto-generates test inputs suitable for checking C to HLS-C conversion errors, while providing high branch coverage for the original C code.

An evaluation of HeteroGen shows that it produces an HLS-compatible version for nine out of ten real-world heterogeneous applications fully automatically, applying up to 438 lines of edits to produce an HLS version 1.63× faster than the original version.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Error handling and recovery**; • **Computer systems organization → Heterogeneous**.

KEYWORDS

Heterogeneous applications, test generation, program repair

ACM Reference Format:
Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. 2022. 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), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507748

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507748

1 INTRODUCTION

Heterogeneous computing, which builds on domain-specific accelerators represented by FPGA and GPU, has shown great promise in performance improvement and energy efficiency. The past decade has seen a proliferation of highly-integrated heterogeneous computing tools and platforms, such as Intel's CPU+FPGA multi-chip packages [20, 48], Amazon's FPGA-enabled AWS cloud [8], Microsoft's FPGA-enabled Azure cloud [46], as well as Google's TPU cluster [24]. However, despite their wide availability in industry, these platforms are notoriously difficult to program. As such, developing heterogeneous applications[1] can only be done by a small handful of programmers with deep hardware design expertise.

There have been continuous efforts to lower the bar for programming with accelerators. Among these efforts, the most successful one is *high-level synthesis* (HLS) [15]. HLS raises the level of programming abstraction from hardware description languages (such as Verilog) to C/C++ dialects (such as HLS-C), enabling C/C++ developers to easily program for FPGAs. Although HLS significantly simplifies accelerator programming, it still requires a substantial amount of manual rewriting from developers to turn a regular C/C++ program into its HLS counterpart, because HLS supports only a subset of the C/C++ language constructs. As a result, developers often have to battle a sea of HLS compatibility errors (*e.g.*, "ERROR: [XFORM 202-876] Synthesizability check failed: recursive functions are not supported.") before their code can even compile. In fact, a quick search on Xilinx's FPGA HLS development forum [59] brought us thousands of Q&A posts on how to fix HLS compatibility errors; a study of these posts (§5.1) reveals numerous confusions and challenges real-world developers have experienced in restructuring application logic to make their code HLS compatible.

Furthermore, to reap performance benefits, developers must explicitly optimize their HLS programs by taking into account low (microarchitectural) level hardware details. For example, they often need to express microprocessor level parallelism (*e.g.*, with pragmas) in their HLS-C code. Configurations such as pipeline depth often need to be specified as well. Despite various attempts [14, 53, 53, 60] to simplify manual rewriting (*e.g.*, by providing libraries for common data structures [61] or that can substitute recursions [53]), it is still a tedious and error-prone process, precluding practical adoption of HLS and the hardware accelerators it supports.

HeteroGen. This paper presents HeteroGen, an automated tool that takes as input a regular C/C++ program and produces its HLS-C counterpart *without involving any human developer in the loop.*

[1] In this paper, term "heterogeneous application" refers to programs that consist of *host* code, which runs on a CPU, and *kernel* code, which can be offloaded to an accelerator such as FPGA.

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.

Realizing these benefits requires two major steps: (1) automatically resolving compatibility issues and (2) generating code with better performance. Our key insight is that these two steps can be organically combined in an iterative code edit process, known in the software engineering community as *evolutionary program repair*. At the heart of HeteroGen is a search-based repair process with a unified objective function that aims to simultaneously reduce the number of compatibility errors and improve performance. Starting from the input C program, each iteration of the process applies a number of edits to the "current" program version, with the goal to generate a new version optimized for both compatibility and performance. This process terminates when a user-specified time budget is reached.

Essentially, fixing compatibility errors is a *hard constraint*—our tool always attempts edits to ensure HLS compatibility. Improving performance is a *soft constraint*—each iteration applies the edits with the largest performance potential when multiple repair candidates are available, although the chosen edits may not lead to optimal performance improvement. In other words, HeteroGen puts higher priority on HLS compatibility and test behavior preservation (although in most cases, edits HeteroGen ends up choosing edits that both improve performance and fix compatibility issues). When the budget runs out, HeteroGen terminates the search, either producing an error-free version with better performance in most cases, or reporting an incomplete version with generated tests to guide the remaining manual edits. If all errors are fixed before the budget runs out, HeteroGen still continues the search to apply performance-improving edits.

Challenges. While search-based program repair has been extensively studied [29, 31, 57], naïvely applying existing techniques would not work in our setting. Existing techniques assume that each program variant can quickly compile and run; their success builds on a "trial-and-error" approach that often involves thousands or even millions of repetitive edits and runs. However, compiling an HLS-program requires an expensive synthesis/simulation process, which takes orders-of-magnitude longer (*e.g.,* minutes to hours) than compiling a regular C program. As a result, existing techniques are prohibitively expensive when adopted directly to repair HLS-C programs.

Rationale for testing as opposed to static verification. Applying an edit may alter program semantics and hence the resulting program must be validated to guarantee correctness. HeteroGen uses test generation and execution to check behavior preservation, as opposed to static verification for two reasons. First, there exists no validating HLS compiler that guarantees semantic equivalence between C programs and HLS-C versions. When a programmer transforms C to HLS-C, currently, there is no way to verify equivalence other than executing both versions with test inputs. Second, it is theoretically impossible to build such a validating compiler between C and HLS-C because all dynamic and unbounded data structures must be finitized to static-size implementations with finite resources in FGPA. Therefore, in the presence of finitization necessity, it is infeasible to ensure semantic equivalence between resource-unbounded SW implementation against resource-finite hardware implementation.

To overcome these challenges, we develop three novel techniques as elaborated below:

1. Automated Test Generation. Determining whether an iteration produces a better program candidate requires understanding whether this candidate (1) preserves the behavior of, and/or (2) outperforms the current version. This also requires tests to execute programs and measure their functional and performance results. For example, if a candidate passes all tests and outperforms the current version, the candidate is *accepted* by the search algorithm as the new "current" for further exploration. However, most programs do not come with tests and it is unrealistic to require tests from developers.

To solve this problem, HeteroGen uses a novel test generation technique to automatically generate tests. Inspired by automated fuzz testing [7], our technique monitors the branch coverage of the original C code to find inputs that diversify branch executions. It stores the intermediate program states in the input C program, mutates them to generate new kernel inputs, and ensures that the mutated inputs are *type-valid* for HLS. With the generated tests, HeteroGen executes both the input C program and each generated HLS version, using result differentiation as a fitness function [57].

2. Dependence-Guided Search Space Pruning. 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. Applying these patterns not only repairs compatibility errors but also leads to improved performance. We encode them into parameterized *repair templates* at the level of abstract syntax trees, providing strong guidance in the search regarding what to edit.

Based on the observation that these templates often exhibit dependence [40, 41], HeteroGen explicitly models the dependence and precedence among these templates. Such dependence information is used to expedite the exploration of applicable repairs using evolutionary algorithms [57]. For example, suppose that there are multiple ways of applying repairs but some repairs depend on others (*e.g.,* repair B depends on A, D depends on B or C), an evolutionary algorithm can enumerate the space of applicable repairs in an order described by their dependence (*e.g.,* {A, C, AB, ABD, …}).

3. Early Candidate Rejection Using Coding Styles. 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.* For example, when inserting an `HLS unroll` pragma to enable loop optimization, HeteroGen invokes an LLVM front-end for HLS to check whether such a pragma appears only within a loop body. In doing so, HeteroGen quickly rejects invalid repairs before they get compiled.

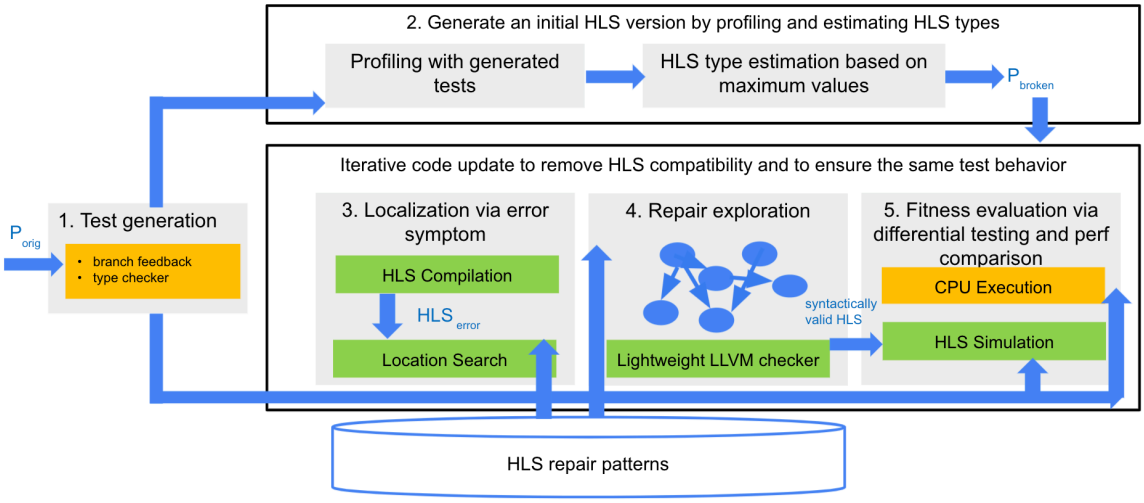


Figure 1: HeteroGen takes as input an original kernel program (P_{orig}). It auto-generates test inputs for P_{orig}, and an initial version P_{broken} with estimated HLS types. Next, it finds repair locations based on an HLS error symptom, explores the space of applicable repairs based on fix patterns, and evaluates behavior preservation via differential testing.

Results. We have evaluated HeteroGen on ten publicly available real-world applications. Out of the ten programs, HeteroGen managed to produce an HLS-compatible version fully automatically for nine. HeteroGen repaired 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. All of the repaired programs produce identical input-output behavior under the generated tests.

We have compared HeteroGen with three alternatives (1) HeteroGen without the coding style checker, (2) HeteroGen without dependence-based repair exploration, and (3) prior work HeteroRefactor [33]. Dependence-guide search expedites the iterative process by 35× compared to (2). The lightweight LLVM checker avoids unnecessary HLS compilation and simulation, leading to an overall of 4× speedup compared to (1). HeteroGen achieves 5× transpilation success compared to (3).

We provide access to artifacts of HeteroGen at https://github.com/UCLA-SEAL/HeteroGen.

2 BACKGROUND

Code Rewriting for HLS. HLS for FPGA [15, 17] has raised the abstraction of hardware development by automatically generating register-transfer level (RTL) descriptions from code written in C-like dialects for HLS. However, a developer must perform a substantial amount of manual rewriting before it can run on an FPGA chip. While the instruction set architecture (ISA) for CPU defines integer arithmetics at 32 bits, in FPGA, individual bitwidths could be programmed [33]. At a high level, regular software developers often allocate variables with a size large enough for all possible input values. Such a practice may lead to wasted on-chip resources, impacting the maximum operating frequency, parallelism, and power consumption. Thus, developers must *finitize* the bitwidth manually to achieve resource efficiency. For example, in modern ML applications where on-chip resource usage is input-dependent, determining an optimized bitwidth is a daunting task.

HLS dialect languages are a strict subset of C/C++ and certain constructs or coding styles are unsupported [50]. A developer must manually restructure the program to make the computation logic *synthesizable* at the hardware level. This is also a difficult task due to the large discrepancy between C/C++ and a HLS C-dialect. During code conversion, there are four primary causes of incompatibility errors:

First, FPGA has no capabilities for managing data structures of an unbounded size. Thus, function calls to dynamic memory management such as `malloc` and `free` must be replaced by pre-allocated static arrays of a conservatively large size. Similarly, recursions must be transformed to loop-based iterations because all required hardware resources need to be pre-allocated. Second, HLS compilers support fewer data types than C/C++. For example, a `long double` type is not synthesizable and must be converted to a HLS floating type such as `fpga_float<8,71>`. Third, pointers are strictly forbidden in HLS, except for special-purpose pointers that are used to express hardware interfaces. Thus, a developer must manually eliminate pointer declarations and usages.

Finally, FPGA provides inherent hardware-level parallelism through pipelining of different computation stages or by duplicating processing elements. As such, developers must manually insert a number of pragmas (*i.e.,* pre-processor directives) to specify how computation pipelining and duplication should be implemented. For example, `#pragma HLS array_partition` partitions a large array into smaller arrays to allow for simultaneous operations. A significant number of HLS incompatibility issues arises in specifying such pragmas. For example, when a developer defines an array `A` with 13 elements but inserts `#pragma HLS array_partition factor=4`, an HLS compiler may produce the following warning "ERROR: [XFORM-711] Array A failed dataflow checking," because 13 is not a multiple of 4.

```
1  void init(Node **root) {
2    *root = (Node *)malloc(sizeof(Node));}
3  ...
4  void traverse(Node *curr) {
5  ...
6    - int ret = visit(curr->val);
7    + fpga_uint<7> ret = visit(curr->val);
8    traverse(curr->left);
9    traverse(curr->right);}
10
11 float kernel(float input[]) {
12   init(root);
13   ...
14   traverse(root);}
```

(a) Initial HLS-C Version with HLS data types

```
1  Node Node_arr[NODE_ARR_SIZE];
2  void init( Node_ptr  *root) {
3    *root =  (Node_ptr)  Node_malloc (sizeof(Node))
           ; }
4  ...
5  void traverse_converted( Node_ptr  curr)
6  {..}
7
8  float kernel(float input[]) {
9    init(root);
10   ...
11   traverse(root);}
```

(b) HeteroGen converted code

```
1  void  traverse_converted (Node_ptr curr){
2  void  stack<context> s(STACK_SIZE);
3
4  while(s.empty()=false){
5    context c = s.pop();
6    goto c.location;
7
8    L0:
9      c.location = L1;
10     s.push(c);
11     s.push({curr: Node_arr[curr].left});
12     continue;
13   L1: ...}}
```

(c) Stack-based implementation of `traverse`

Figure 2: Working example of HeteroGen.

Automated Program Repair. Search-based program repair [21, 42, 57] has shown promises for patch generation and repair. Without loss of generality, the overall procedure of evolutionary program repair [57] can be described as follows. Starting from a base program that fails (*i.e.,* does not satisfy a repair success oracle), program repair generates a new program variant in each iteration by applying code edits to a current program variant. Next, the program variants are evaluated using a fitness function. This process is repeated until a program variant that passes all tests is found.

These techniques build on two fundamental assumptions: (1) the program under repair can be quickly compiled and executed (*e.g.,* milliseconds) and (2) test cases (or an alternative oracle) are available to assess the fitness of a repair. Unfortunately, neither of these assumptions holds for heterogeneous applications, because an HLS program takes minutes and hours to compile and test cases are unavailable in most cases.

3 HETEROGEN OVERVIEW AND EXAMPLE

System Architecture. HeteroGen consists of five components, as shown in Figure 1: (1) test input generation, (2) initial HLS version generation, (3) identification of repair locations, (4) repair space exploration, and (5) fitness evaluation. Steps (3) to (5) repeat as a part of iterative repair constrained by a given time limit.

HeteroGen takes as input an original C/C++ program `P_orig`. First, it generates test inputs to maximize branch coverage in `P_orig`. Second, it runs `P_orig` with test inputs and constructs an initial HLS version `P_broken` by estimating the maximum size of HLS data types for individual input variables. Then by compiling `P_broken` with a HLS compiler, it finds whether any HLS compatibility error exists. Guided by these error symptoms, it explores applicable edits and terminates the repair process if the variant corrects all HLS compatibility errors, preserves identical test behavior, and yields better performance than the original version.

Working Example. Consider a binary tree program (128 LOC) shown in Figure 2a. Suppose Alice would like to synthesize this entire program on FPGA using an HLS compiler. The HLS compiler would report three error messages indicating that the pointer usage for dynamic memory management (lines 2) and the recursion (lines 8-9) are not supported, such as "ERROR: [XFORM 202-876] Synthesizability check failed: recursive functions are not supported." and "ERROR: [SYNCHK 200-61] unsupported memory access on variable curr which is (or contains) an array with unknown size at compile time."

Without HeteroGen, Alice would have to manually rewrite code to use static array accesses and iterations instead. Such manual refactoring would take significant efforts (*i.e.,an extra of 196 LOC*), to produce a working version of 324 LOC. Unfortunately, since no test inputs are available, Alice can only roughly verify the functionality of the rewritten program with handcrafted or random inputs.

HeteroGen solves this painful problem by automating the code conversion process. Based on the original C program, HeteroGen first generates 1,800 test inputs of floating point arrays in 50 minutes because the `kernel` function accepts data of the floating point type as input in line 11 of Figure 2a. These inputs achieve full branch coverage. Next, HeteroGen profiles this program with the generated tests and finds that the maximum value for the local variable `ret` is 83. HeteroGen updates its data type to `fpga_uint<7>` to create an HLS version to begin with. Without any manual effort from Alice, HeteroGen uses the HLS error messages that arise during HLS compilation to search for applicable fixes. It then applies, (1) the *array replacement* edit (highlighted in red) to replace `malloc` with array-based memory accesses `Node_malloc` in line 3 of Figure 2b; (2) the *pointer removal* edit (highlighted in blue) to replace pointers `Node *` to array indices `Node_ptr`; (3) the *stack replacement* edit (highlighted in green) to replace recursions with iterations based on `stack` in line 2 of Figure 2c; and (4) the *array resizing* edit to experiment with different array sizes in line 1 of Figure 2b and line 2 of Figure 2c. After many iterations of applying other applicable edits, HeteroGen generates a final version with 464 LOC that does not have errors and outperforms the original C program.

Caveat and Usage Scenario. HeteroGen takes the kernel functions in a C/C++ application as input and generates their equivalent HLS-C versions. In other words, HeteroGen does not reinvent the wheel of finding performance bottlenecks and code to be offloaded to HW accelerators and instead assumes that kernel code to be transformed is specified. Many existing tools such as *e.g.,* valgrind [55] could identify kernel code by profiling an application.

Porting kernel functions to FPGAs involves error fixing and parallelization, both of which are challenging tasks. HeteroGen focuses on error fixing; although it can also lead to increased efficiency by applying performance-improving edits. HeteroGen does not perform auto-parallelization and -tuning that often require

Table 1: Example HLS compatibility errors.

Type	ID	Error Symptom	Repair
Dynamic Data Structures	729976 [1]	Allocating an array with unknown size leads to "ERROR: Dynamic memory allocation is not supported"	Specify the array size
Unsupported Data Types	752508 [2]	The `long double` variable leads to "ERROR: Call of overloaded 'pow()' is ambiguous"	Type transformation, followed by explicit type casting and operator overloading
Dataflow Optimization	595161 [3]	Inserting `dataflow` pragma leads to "ERROR: Argument 'data' failed dataflow checking"	Pragma exploration
Loop Parallelization	721719 [4]	Inserting `dataflow` pragma and `unroll` pragma fails the pre-synthesis	Pragma exploration
Struct and Union	1117215 [5]	Struct leads to "ERROR: Argument 'this' has an unsynthesizable struct type"	Insert an explict constructor and make the connecting stream static
Top Function	810885 [6]	Incorrect configuration leads to "ERROR: Cannot find the top function in the design"	Configuration Exploration

Algorithm 1: HeteroGen's test generation.

```
   Input: a program P with kernel function K, sample inputs s for P
1  Function getKernelSeed (P,s)
2      return arguments sent to K in P.run(s)
3  Function fuzzing(P, s)
4      ks = getKernelSeed (P,s);
5      type = argument's type in K;
6      inputs.append(ks);
7      while given time budget do
8          test_case_list = Mutation(inputs.pop(),type);
9          foreach test_case ∈ test_case_list do
10             feedback = K.run (test_case);
11             if NewCov(feedback) then
                   inputs.append(test_case) ;
12         end
13     end
```

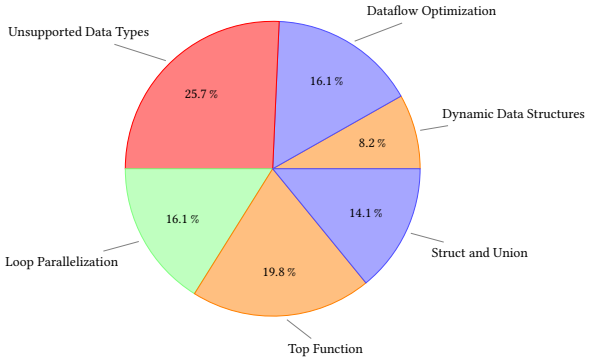


Figure 3: HLS compatibility error types in Xilinx forum.

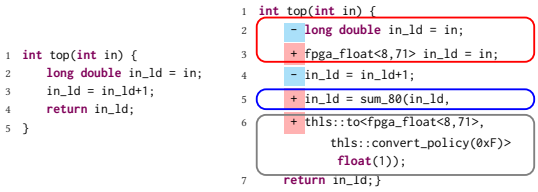
algorithmic redesign with intimate knowledge of hardware. In real-world development, we envision developers can use HeteroGen in a "change-and-fix" loop where in each iteration they can apply algorithmic and/or hardware-specific changes first to produce a "draft" and then use HeteroGen to generate a compatible and more performant version from that draft.

4 TEST GENERATION

Our goal is to find an HLS-C program that is able to pass HLS compilation and that yields equivalent behavior between CPU and FPGA. In practice, programmers perform HLS differential testing between CPU and FPGA using handcrafted or random inputs. Existing test generation techniques, such as AFL [7], are not directly applicable to heterogeneous applications for two reasons: (1) it targets the end-to-end application as opposed to the kernel code only, while the goal of HLS differential testing is to compare input-output behavior of the kernel under CPU vs. FPGA; (2) the input mutation strategy used in existing techniques does not consider HLS data type compatibility; as such, when the newly generated kernel inputs are not compatible with HLS-data types, most inputs would fail at the kernel entry point without exercising any kernel logic further.

Algorithm 1 outlines HeteroGen's input generation strategy. It starts the test generation process with an original program that consists of both host and kernel code as well as an initial set of inputs that are randomly generated. Function `getKernelSeed` captures the actual value states at the entry of the kernel function, *e.g.*, actual values passed to the kernel function. Such intermediate state is then used as the seed input for kernel input generation. The insight behind extracting a seed rather than handcrafting a random seed is that such intermediate states are ensured to be valid, leading to improved fuzzing efficiency. Next, HeteroGen analyzes the argument types used in the kernel function, and inserts additional type checkers in the fuzzing loop, as shown in line 5 and line 8. In other words, HeteroGen generates hard-to-reach corner case inputs with valid HLS data types. HeteroGen then executes the program with every newly generated input in line 10. During this execution, HeteroGen collects feedback which indicates code coverage information. If an execution results in new code coverage (*i.e.*,`NewCov`), then the corresponding input is added to the input queue for further fuzzing in line 11.

Initial HLS-C Version Generation. HeteroGen generates the initial HLS version for a C/C++ application by estimating the HLS data types in the kernel code. Similar to [33], HeteroGen profiles the kernel code to keep track of the maximum values for intermediate variables with the generated tests. For example in Figure 2a, a programmer uses a 32-bit integer for the variable `ret` by default in line 6, which is a higher bitwidth than what is actually necessary. HeteroGen finds that it has a max value of 83—it then only needs 7 bits instead of 32 bits. It parses the program's AST, identifies the variable declaration node for `ret`, and then modifies the corresponding type as shown in line 7 in Figure 2a.



(a) Original Code (b) Repaired Code

Figure 4: Example repair for unsupported data type.

5 AUTO-REPAIRING COMPATIBILITY ERRORS

After generating the initial version with estimated HLS types, HeteroGen automatically evolves the program to succeed HLS compilation while finding the code variant with best performance among all applicable edits. In particular, inspired by automated repair, we develop novel techniques specifically to address the need of semantics preservation and performance optimization in HLS where a naïve "trial-and-error" approach is prohibitively expensive. For this, we design fix patterns that are specific to common error types found in our study of real-world HLS compatibility errors. We expedite the search space exploration by both *reducing the repair attempts* using dependence-based search and *reducing the HLS compilation time for each repair* using coding style checks.

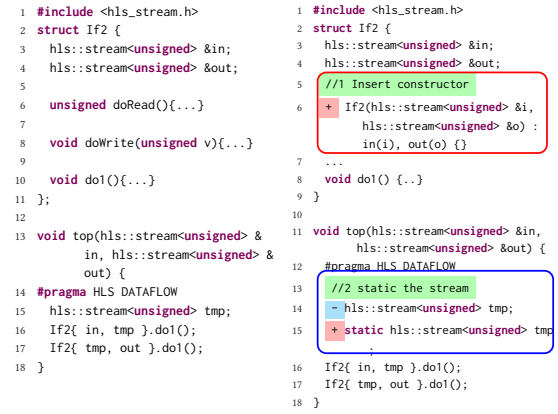
5.1 A Study of HLS Compatibility Errors

To understand real-world HLS compatibility errors, we collected 1,000 posts from Xilinx's HLS Q&A forum using a keyword-based search with the search term "high level synthesis error" and "C synthesis error." We carefully examined the accepted answers and associated comments to understand the root causes of underlying HLS incompatibilities and summarize their repair solutions. We then distilled and grouped these root causes into six categories, each reflecting an underlying HLS incompatibility issue: dynamic data structures, unsupported data types, dataflow optimization, loop parallelization, struct and union, and *top functions—i.e.,* each hardware design has a *top function* specifying its module entry point and inserted pragmas specify the module's configuration interface.

The pie chart in Figure 3 illustrates the proportion of these six error types. The most frequent source of HLS incompatibility is unsupported data types, which accounts for a quarter of total cases. Such errors occur when eliminating pointers or when adding support for custom HLS data types. Configuration-related top function errors, dataflow optimization, and loop parallelization are other major sources of HLS incompatibility, as it requires deep hardware platform knowledge to specify appropriate pragmas. 14% of HLS incompatibilties are caused by the use of struct and union. Last but not least, dynamic data structures contribute to 8%, due to the use of `malloc`, `free` and recursive functions.

Table 1 summarizes each HLS incompatibility type, a representative example post ID, its error symptom, and corresponding repair edits. Examples for each type are shown below.

- *Dynamic Data Structures*: Post No. 729976 [1] shows an example where a developer attempts to allocate an array `MY_DATA line_buf_a[WIDTH][cols]` where the value of `cols` is unknown at compile time. Thus, HLS does not know the exact amount of hardware resources to allocate, leading to a failed synthesis with two errors "ERROR [SYNCHK-31] dynamic memory allocation/deallocation is not supported" and "ERROR [SYNCHK-61] unsupported memory access on variable line_buf_a." To correct these errors, the array size must be declared as a constant after experimentation with different array sizes. *Performance implication*: fixing these errors reduces communication frequency between CPU and FPGA.
- *Unsupported Data Types*: Post No. 752508 [2] presents an example of unsupported data type `long double`. Initially, a trigonometric function is declared with `long double` variables, leading to arithmetic operator overloading errors. Figure 4 demonstrates code repairs to fix such errors. Lines 2-3 in Figure 4b replace a `long double` type to a `float` type with a custom bitwidth `fpga_float<8,71>`. Line 6 explicitly performs this type casting by changing a 32-bit integer to this float type, because implicit type casting is not well supported in HLS. Line 5 manually overloads a corresponding custom arithmetic operation for this type. *Performance implication*: customizing data types reduces resource consumption, which directly impacts the parallelism level and operating frequency.
- *Dataflow Optimization*: HLS developers may insert `#pragma HLS dataflow` to enable task-level pipelining, allowing overlap and simultaneous execution of involved tasks. In Post No. 595161 [3], a sub-function `my_func(char data[128])` is called twice in `top_function`, inducing "ERROR: Array 'data.0' failed dataflow checking," because the same input `data` is passed to two simultaneous `my_func` invocations. Such dataflow optimization errors could be fixed by segmenting the original input data into multiple small arrays of different sizes to enable simultaneous, independent computation. *Performance implication*: segmenting data creates finer-grained tasks, leading to increased degree of parallelism.
- *Loop Parallelization*: Similar to dataflow optimization, loop-optimization specific pragmas can induce HLS errors. In post No.721719 [4], "ERROR [HLS-70] Pre-synthesis failed" occurs after inserting an `unroll` pragma in the loop body. However, this error occurs only with an unrolling factor of 50 or more



(a) Original Code (b) Repaired Code

Figure 5: Example repair for unsynthesizable struct.

Table 2: Parameterized edits for each error type.

Type	Example Parameterized Edits
Dynamic Data Structures	`array_static($a1:arr,$i1:int)`, `insert($a1:arr,$d1:dyn)`, `resize($a1:arr)`, `stack_trans($d1:dyn)`, etc.
Unsupported Data Types	`pointer($v1:ptr)`, `type_trans($v1:var)`, `array_static($a1:arr)`, `type_casting($v1:var)`, etc.
Dataflow Optimization	`delete($p1:pragma,$f1:func)`, `move($p1:pragma,$f1:func)`, `insert($p1:pragma,$f1:func)`, etc.
Loop Parallelization	`index_static($l1:loop)`, `mem_reset($l1:loop)`, `init($l1:loop)`, `explore($p1:pragma,$l1:loop)`, etc.
Struct and Union	`constructor($s1:struct)`, `flatten($s1:struct)`, `stream_static($f1:stream,$s1:struct)`, `inst_static($s1:struct,$v1:name)`, `pointer($s1:struct)`, etc.
Top Function	`delete($p1:pragma,$f1:func)`, `move($p1:pragma,$f1:func)`, `insert($p1:pragma,$f1:func)`, etc.

```
1 static bool is_recursion(FunctionDeclaration *func){
2       auto ref = isFunctionRefExp(i);
3       auto ref_func = ref->getAssociatedFunctionDeclaration();
4       auto def_ref_func = ref_func->get_definingDeclaration();
5       if(def_ref_func == func)
6         return true;}
```

Figure 6: Repair location for recursion.

because of two interacting pragmas: a pre-existing `dataflow` pragma and the `unroll` pragma with factor 50. This error could be removed by setting up an explicit total number of iterations performed by a loop, making all indexed items static, and exploring combinations of pragma `dataflow` with a different tripcount. *Performance implication*: unrolling a loop appropriately leads to parallelization and performance improvement.

- *Struct and Union*: To use structs and unions in HLS, a developer must declare supporting hardware level implementations accordingly. Post No. 1117215 [5] demonstrates an error caused by unsynthesizable structs shown in Figure 5. Using two `struct` instances in lines 16-17 in Figure 5a is not supported in HLS, because there are no corresponding constructor and data transfer implementations at the hardware level. To fix this error, a developer must declare an explicit constructor in line 6 in Figure 5b and an associated static streaming function `tmp` in line 15 in Figure 5b. *Performance implication*: supporting structs and unions also reduces fallback and communication.
- *Top Function.*: A top function is a module entry point (*i.e.,* a hardware interface), and an error may occur when its configuration such as a clock frequency or a device name is incorrect, has an incorrect data path, or misspells a top function name, *e.g.,* Post No. 810885 [6].

Key Takeaway. Common fix patterns extracted from user posts can provide clear guidance in the search process so that each iteration can focus on meaningful edits rather than trying out random edits most of which are guaranteed not to work. In fact, all but one programs in our experiments were successfully fixed with these patterns guiding the search process. Five of these six patterns can also improve performance. As a result, most of our edits can lead to more efficient programs as well. How to parameterize these patterns and encode their dependences will be discussed in the following sections.

5.2 Repair Localization

Spectrum-based fault localization is a commonly used technique in locating where to apply repairs [29, 35, 57]. HeteroGen designs an HLS-specific repair localization method based on HLS compiler error messages. The key insight here is that *HLS compiler error messages often provide a crucial hint on which language constructs must be modified to make it HLS compatible.* For example, based on an error message "ERROR: [XFORM 202-876] Synthesizability check failed: recursive functions are not supported," we can locate a recursive function whose invocation target name is the same with its defining declaration (line 5 in Figure 6). HeteroGen is equipped with an error-type specific localization. Currently, HeteroGen classifies each HLS error message to one of the six types described in §5.1 by extracting keywords such as "recursion," "dataflow," or "struct," *etc.* Then it finds potential repair locations for each error type. In HeteroGen, this repair localization module is designed for extensibility—for a new HLS error type, a user can add a new corresponding repair localization module.

5.3 Repair Exploration

Given a heterogeneous application, the HLS compilation process involves a series of operations: scheduling, resource allocation, binding, and mapping, etc. This process, together with the simulation, can take several minutes to hours, depending on kernel logic complexity. Such high latency makes HLS not suitable for traditional evolutionary repair where, after each repair attempt, a compiler is invoked and the compiled program is executed with given inputs. Below we describe how HeteroGen reduces automated repair time.

HLS Coding Style Validity. HLS has a phased, top-down compilation and execution flow. Our observation is that we can always safely terminate the compilation for a program that does not adhere to HLS coding styles. Such style checking can be performed without setting up a time-consuming HLS environment. Thus HeteroGen leverages a lightweight LLVM frontend specifically for HLS coding style checks before invoking the full HLS compilation process. For example, when inserting an `HLS array_partition` pragma to enable parallel operations on arrays, HeteroGen invokes this checker to ensure this pragma is inserted within the boundaries of the function, where the array variable is defined. Although such LLVM-based checking has non-zero cost, this time is negligible compared to invoking the full HLS compilation process with hardware resource allocation, scheduling, binding, and technology mapping.

Dependence-based Repair Exploration. In prior work, automated program repair leveraged fix patterns extracted from correct reference code [66], bug fix histories [31, 39], or human-written patches [29] and used such patterns to explore the space of repair candidates. In the HLS domain, there is a unique opportunity to draw hints on where to apply repairs based on HLS compiler error messages. For each error message, HeteroGen maps the message

```
1  struct If2 {
2    constructor(If2)
3
4    If2(hls::stream<unsigned> &i,
5    hls::stream<unsigned> &o)
6    : in(i), out(o)
7  ...};
8
9  void top(hls::stream<unsigned> &in, hls::
       stream<unsigned> &out) {
10   stream_static(tmp,If2)
11
12   hls::stream<unsigned> tmp;
13   static hls::stream<unsigned> tmp;
14 If2{ in, tmp }.do1();
15 If2{ tmp, out }.do1();
16 }
```

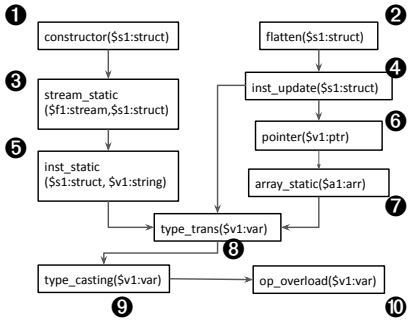
❶ ❸

(a) Parameterized Edits for *Struct and Union*

```
1  struct If2 {
2    flatten(If2)
3    hls::stream<unsigned> &in;
4    hls::stream<unsigned> &out;
5    unsigned doRead() const{
6    unsigned doRead(hls::stream
7     <unsigned> &in)const{
8   ...} ...};
9
10 void top(hls::stream<unsigned> &in, hls::
       stream<unsigned> &out) {
11 hls::stream<unsigned> tmp;
12   inst_update(If2)
13   If2in,tmp.do1();  If2tmp,out.do1();
14   do1(in,tmp);  do1(tmp,out);
15 }
```

❷ ❹

(b) Alternative Parameterized Edits for *Struct and Union*



(c) Dependence and Precedence Structure

Figure 7: Dependence Structure of Parameterized Edits. Red marks additions, and gray marks deletions.

to an error type, corresponding fix patterns, and the dependence relations among constituent repair edits, as shown in Table 2.

HeteroGen encodes the repair operations as parameterized edits whose variable, function, and type names could be concretized to a given context. For example, the HLS error on *struct and union*, the following edits may be needed:

- `constructor($s1:struct)`: insert a constructor of `$s1` if not existed;
- `flatten($s1:struct)`: flatten `$s1` with its standalone variables and methods;
- `stream_static($f1:stream, $s1:struct)`: make a static data transfer streaming function `$f1` for struct `$s1` instances, if exists;
- `inst_static($s1:struct, $v1:string)`: make a static instance of struct `$s1` with an assigned name `$v1`;
- `pointer($s1:struct)`: rewrite pointers in struct `$s1`.

Since a single error could be fixed in multiple ways, we define a set of constituent edits and describe dependence relations of those edits. For example, for the *Struct and Union* error type, ten constituent edits could be defined, as shown in Figure 7. Figure 7a shows the code differences before and after parameterized edits ❶❸ in which we insert an explicit constructor in the struct `If2` followed by making the connection stream `tmp` static. Alternative parameterized edits ❷❹ are shown in Figure 7b in which we flatten the struct `If2` with standalone methods and update all the associated instances.

We summarize the dependence relations among parameterized edits for this error type in Figure 7c. Evolutionary algorithms can use this type-specific dependence structure to enumerate the space of applicable repairs, for example {❶, ❷, ❶❸, ❷❹, ❶❸❺, ❷❹❽,...}. During this search process, HeteroGen records the simulation time for different repair candidates, and finds the variant with the best performance within the search space. If errors of other types arise in a variant, HeteroGen adds this variant to the queue and repeats the dependence-based search until any repair candidate behaves identically with the original program.

Similarly, to fix a *dynamic memory allocation* error, developers may need an `array_static($a1:arr, $i1:int)` edit to declare an array with size `i1`, and such edit naturally requires a subsequent `resize($a1:arr)` edit to experiment with different array sizes for `$a1`. However, these two edits cannot work in a reverse order. HeteroGen extracts edits for removing dynamic memory allocations, pointers, and recursions based on [33] and leverages dependence relations among constituent repair edits to accelerate enumeration of applicable repairs.

Behavior Preservation via Differential Testing. Using a set of tests generated from §4, HeteroGen executes the original C/C++ application on CPU. Then it compares the outcome against the simulation outcome of a heterogeneous variant being constructed. HeteroGen computes the ratio of tests that have identical behavior, and compares the simulation latency of the generated tests between CPU and FPGA. In other words, HeteroGen considers both semantics preservation and performance improvement as a code generation goal.

6 EVALUATION

We evaluate following research questions:

RQ1 How often can HeteroGen produce a heterogeneous application that can guarantee the same behavior with better performance?

RQ2 How efficient and effective is HeteroGen's test input generation for aiding HLS compilation?

RQ3 How efficient is HeteroGen's evolutionary repair?

RQ4 How does HeteroGen's auto-generated version compare to the manual developer version and prior work of HeteroRefactor [33] in terms of performance and code size?

Benchmarks. We evaluated HeteroGen with ten C/C++ applications with FPGA as accelerators, listed in Table 3. They include *eight* microbenchmarks (P1-P8) from prior work [33] or gathered from Xilinx forum, and *two* real-world applications (P9-P10) from the Rosetta benchmark [67]. All of these programs were taken from publicly available sources, and their issues represent real-world programming challenges.

These programs may look small to researchers of pure-software systems, but they are *larger* than other benchmarks on specialized hardware accelerator synthesis. Our evaluation subjects have up to 465LOC, compared to 200LOC for MachSuite[49] and 100LOC for Intel's t2sp [22]. The complexity of HeteroGen's transpilation depends on types of HLS compatibility errors, not the code size of

Table 3: Subjects and overall results.

ID	Subject	HLS Compatibility	Improved Performance?
P1	signal transmission	✓	×
P2	arithmetic computation	✓	✓
P3	merge sort	✓	✓
P4	image processing	✓	✓
P5	graph traversal	✓	✓
P6	matrix multiplication	✓	✓
P7	bubble sort	✓	✓
P8	linked list	✓	✓
P9	face detection	✓	✓
P10	digit recognition	✓	✓

the original program. A bigger program could be handled as long as the compatibility error is one of the six supported types.

Experimental Environment. All experiments were conducted on a machine with Intel(R) Core(TM) i7-8750H 2.20GHz CPU and 16 GB of RAM running Ubuntu 18.04. The test generation was built on AFL version 2.52b [7]. The code transformation was implemented based on LLVM version 8.0.0 [37]. The converted programs were targeted at a Xilinx Virtex UltraScale+ XCVU9P FPGA on a VCU1525 Reconfigurable Acceleration Platform. Latency was reported by the FPGA simulator.

6.1 Program Conversion Effectiveness

We assess HeteroGen's efficacy by inspecting if the produced HLS-C program achieves both *HLS compatibility* and *better performance*. We empirically set three hours as the terminating time limit. As shown in Table 3, HeteroGen has successfully fixed all HLS compatibility errors in all programs, and nine of them outperform the original programs. Because HeteroGen's fix patterns are drawn from real-world HLS compatibility fixes, by construction, it produces HLS-compatible code, although some fixes may not improve performance. When multiple repair candidates are applicable to fix the original program, HeteroGen produces the most efficient version. After a careful investigation on P2-P10, we conclude that HeteroGen realizes performance benefits primarily through exploring loop- and array-related pragmas to enable parallelization. For P1, however, the program transforms 3-dimensional RGB signals to YUV signals via basic arithmetics without any loops or arrays. As such, HeteroGen could not perform any performance-improving edits.

6.2 Test Generation

We run test generation for each subject with a random seed and manually terminate the fuzzing process until AFL's `process timing` indicator shows that 30 minutes have passed since exercising the `last new path`. In other words, we generate new tests until branch coverage is no longer increasing significantly despite new input generation. We repeat the process three times and report the average numbers of generated tests, execution time, and corresponding branch coverage in Table 4 (HeteroGen in column HG). In summary, the generated 2,437 (average) tests cover 97% branches in our subjects. This is a significant improvement because not all subjects come with tests and pre-existing tests reach only 36% branch coverage.

Table 4: Generated tests.

Subject	HG # of Tests	HG Time (mins)	HG Cov.	Existing # of Tests	Existing Cov.
P1	27	35	100%	N/A	N/A
P2	6,930	50	100%	N/A	N/A
P3	1,800	50	100%	10	25%
P4	47	55	100%	N/A	N/A
P5	38	41	100%	10	40%
P6	14,896	35	100%	4	33%
P7	399	35	100%	N/A	N/A
P8	54	50	100%	N/A	N/A
P9	43	84	70%	1	15%
P10	133	67	100%	11	70%

```
1 void traverse(Node_ptr curr)
2 {
3   traverse(Node_arr[curr].left);
4   traverse(Node_arr[curr].right);
5 }
```

(a) Recursive program

```
1 void traverse_converted(Node_ptr curr)
2 { stack<context> s(1024);
3  stack<context> s(2048);
4 while (!s.empty()){...} }
```

(b) Stack-based repair

Figure 8: Red marks the repair with the generated tests, while blue marks the repair with pre-existing tests only.

For the programs that come with existing test cases, we run HeteroGen with both existing tests and generated tests. For P3 [33], HeteroGen transforms the recursive traversal in lines 2-3 of Figure 8a to a stack-based implementation. With pre-existing tests, HeteroGen initially sets the stack size as 1024; however, after generating more tests, 44% of the tests produce outcomes different from CPU. Finally, after experimentation with a different stack size and setting it to 2048, all tests produce identical results between CPU and FPGA. This demonstrates the *absolute requisite* of incorporating automated test generation in converting C programs to HLS-C variants to ensure behavior preservation.

6.3 Speedup for Repair Process

HeteroGen leverages a two-fold approach to expedite the repair process: dependency-based exploration to reduce the number of repair attempts and coding style check to reduce latency. We conduct an ablation study to evaluate the benefit of each optimization in isolation. For this, we create two alternative versions as baselines:

- WithoutChecker is a downgraded version of HeteroGen that invokes the full HLS compilation process in each repair attempt without using an LLVM-based HLS style checker.
- WithoutDependence is a downgraded version of HeteroGen that chooses any candidate edit in a random order. This version still invokes an LLVM-based HLS style checker.

To assess speedup enabled by dependence-based exploration, Figure 9 shows the wall-clock time of the same repair tasks for HeteroGen and WithoutDependence. HeteroGen is up to 35X faster than WithoutDependence. Multiple coordinated edits are necessary for repairing HLS compatibility errors. HeteroGen takes the advantage of dependence relations to accelerate enumeration of applicable repairs, while WithoutDependence applies random edits in each iteration, leading to a much larger search space. For example, the naïve probability of selecting ❸, given that ❶ is already selected, in Figure 7c is 10% (= 1/10) in WithoutDependence. In this case, HeteroGen applies ❸ after ❶ based on dependence.

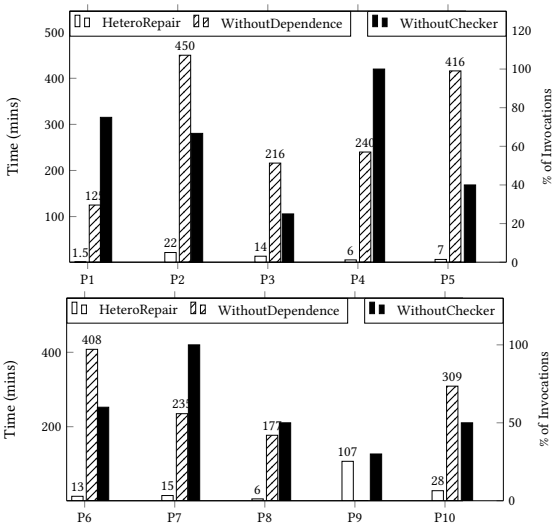


Figure 9: Time and HLS invocations. WithoutDependence fails to achieve HLS compatibility within 12 hours for P9.

To assess speedup enabled by LLVM-based HLS coding style checks, we compare HeteroGen with a downgraded version WithoutChecker that invokes the full HLS compilation for each repair attempt. We report the number of HLS tool chain invocations. In Figure 9, the black bars show the percentages of invoked HLS processes in all repair attempts for each subject. In P3, HeteroGen can obviate the need of invoking the full HLS tool chain by 75%, which results in a 4× speedup. HeteroGen achieves such a speedup by checking HLS-coding styles first, and invokes the subsequent HLS process (*e.g.,* hardware resource allocation, scheduling, technology mapping and binding, etc.) only if the candidate repair conforms to HLS coding styles. This early termination saves time but does not sacrifice HeteroGen's repair capability, because most HLS compatibility errors could reliably be reported in the beginning phase of hardware synthesis.

6.4 Comparison with Human Generated Programs and HeteroRefactor

Table 5 reports the comparison between HeteroGen (HG) and prior work HeteroRefactor [33] (HR). The human-generated versions are either from the accepted answers of online posts, or are shipped with the Rosetta benchmark [67]. For HeteroRefactor, we ran its publicly available version on the same subject programs.

Code Edit. We measure the size of code edits by calculating the number of added lines with respect to the total lines of code in the original program.

First, we note that both HeteroGen's generated HLS programs and human-generated HLS programs produce identical test behavior between CPU and FPGA for all subject programs. As an example, if we manually port program P9 to FPGA HLS, 3272 line edits are required. Such edits utilize a trace-based memory banking technique to pipeline memory access patterns in the Viola-Jones algorithm. In contrast, HeteroGen applies 144 line edits to produce an HLS version. On average, HeteroGen automates 143 line edits, reducing HLS programming effort.

Table 5: Comparison against manual edits and HeteroRefactor [33].

	Origin	ΔLOC			Runtime (ms)			
ID	LOC	Manual	HR	HG	Origin	Manual	HR	HG
P1	15	78	✗	69	0.21	0.11	✗	0.35
P2	24	8	✗	9	0.96	0.45	✗	0.53
P3	121	276	342	356	1.46	1.09	1.19	1.13
P4	285	136	✗	32	8.4	2.01	✗	3.28
P5	85	144	✗	438	1.68	0.91	✗	1.17
P6	19	25	✗	16	1.13	0.35	✗	0.89
P7	50	45	✗	25	3.6	2.31	✗	2.59
P8	131	156	✗	298	3.46	1.28	3.46	1.79
P9	465	3272	✗	144	101	33	✗	47
P10	117	61	✗	35	24.3	10.5	✗	13.6

Second, when comparing HeteroGen against prior work HeteroRefactor [33], we find that HeteroRefactor works only for P3 and P8 out of 10 programs—20% vs. 100% transpilation success for HeteroRefactor and HeteroGen respectively. In fact, HeteroRefactor 's scope is limited to dynamic data structures, while HeteroGen 's scope includes additional dataflow, loop parallelization, struct and union, unsupported data types, and top functions. Therefore, by definition, HeteroGen has a superior capability in transpiling C to HLS-C than HeteroRefactor.

Performance. We assess the performance improvement by comparing the runtime of (1) the converted program on FPGA; and (2) the original kernel code on CPU. The execution latency is reported by the HLS simulator. On average, HeteroGen's converted versions and the manually crafted versions are 1.63× and 2.43× faster than the original CPU versions, respectively. For P3 and P8, HeteroRefactor 's generated code is 1.53× slower than HeteroGen 's output, because HeteroGen can perform additional types of transformations to improve performance.

HeteroGen does not primarily target performance gains. HeteroGen is implemented in an extensible manner such that it is easy to include new transformation patterns. For example, matrix partitioning [14] transformation could be added to improve performance. HeteroGen provides an infrastructure for code conversion automation, opening up massive opportunities for incorporating such (current and future) patterns.

6.5 Limitations

We discuss the limitations of current HeteroGen as follows.

Restricted Platform. HeteroGen is designed for heterogeneous computing with FPGA only. The integrated repair edits are extracted based on Vivado HLS errors and their corresponding corrections. The key idea of HeteroGen could generalize to other FPGA-based platforms by updating the current set of repair edits.

Over-Estimated Bitwidth. The generated test inputs reflect the value range for each type. Consequently, when finitizing resources, HeteroGen often overestimates the stack size, array size, and bitwidths based on the declared types in the original C program. Such type-based over-estimation could lead to resource waste in the transformed program.

Insufficient Performance-Improving Edits. HeteroGen focuses on legacy code rewriting for HLS compatibility errors. Thus, code transformations for algorithmic redesigns or auto-parallelism to enhance performance are left as future work.

7 RELATED WORK

Heterogeneous Computing with FPGA. Heterogeneous computing delivers superior performance for diverse applications (*e.g.*, machine learning, data analysis and graph processing) [25–27, 32, 38, 54, 62, 63]. Programming complexity control has been a long challenge for the adoption of FPGA acceleration. State of the art techniques for advancing heterogeneous computing fall into four primary categories.

Programming languages and compilers. HLS compilers extend C/C++ with ad hoc annotations to express hardware-level concerns [12, 15]. Calyx [43] is a new intermediate language for generating hardware accelerators. It separates the specification of an accelerator's data path from its execution schedule, and aims to generate desired architecture without resorting to low-level RTL engineering. KLOCs [25] proposes a new heterogeneous memory system, and uses a compiler transformation for Verilog to produce performance optimized code.

Domain-specific ISAs. Domain- or application-specific ISAs [16, 19, 56] provide customization opportunities for general ISAs to reduce storage/control overhead by generating compact code, thereby providing a simple programming environment/flow and making FPGA acceleration accessible.

Hardware abstractions. FPGA hardware abstractions [28, 62] provide systems support for resource management. For example, SYNERGY [32] virtualizes FPGA workloads across a cluster of Altera SoCs and Xilinx FPGAs on Amazon F1. Optimus [38] proposes a hypervisor that supports scalable shared-memory virtualization.

Simulation tools: Various simulation tools are designed for better accuracy and performance estimation [26, 27]. For example, FirePerf [27] enables a set of system-level performance profiling capabilities integrated into the FPGA simulator.

Unlike these works, HeteroGen aims to simplify HLS programming, improving developer productivity and program performance.

Code Rewriting for HLS. Enabling high level synthesis of recursive structures has been a long challenge, because unlike CPU, the address space for each array is separate in FPGA. Thomas et al. [53] provide a C++ template library for supporting recursion in HLS but would require a developer to manually rewrite control statements using lambdas. SynADT [61] is an HLS library for linked lists, binary trees, hash tables, and vectors, and it internally uses arrays and a shared system-wide memory allocator [60]. HeteroRefactor [33] builds the dynamic data structure support, and bitwidth optimization for integers and floating points in HLS programs. Unfortunately, code refactoring is error-prone itself and these tools do not generate tests that can validate functionality. Moreover, these tools are not *automated*—they do not account for compatibility issues and require developers to manually refactor their code.

In contrast, HeteroGen automatically converts a C/C++ program to its equivalent HSL-C variant for FPGA-based heterogeneous computing without *requiring any developer involvement.*

Test Generation. Fuzz testing generates new inputs by mutating previous inputs to expose unseen program behavior and it has been highly effective in revealing various bugs, including correctness bugs [10, 44, 45, 64, 65], security vulnerabilities [11, 18], and performance bugs [58]. One important angle to push test generation towards hard-to-reach corners or specific error types is to encode diverse feedback information as a fuzzing guidance metric. For example, while AFL [7] mutates a seed input to maximize cumulative branch coverage, MemLock [58] uses memory consumption as performance-side feedback to detect abnormal memory behavior. HeteroGen takes a similar approach. In addition to monitoring branch coverage, HeteroGen inserts a type checker by analyzing the arguments of kernel code and uses such additional feedback. Thus, generated inputs can better serve the purpose of driving the program execution to a deep path.

Superoptimizers. Superoptimizers use a stochastic process to generate instruction sequences with better performance. Churchill et al. [13] propose a new architecture for superoptimizers by incorporating a fully sound verification technique to ensure correctness and a bounded verification technique to guide the search to optimized code. STOKE [52] formulates the loop-free binary superoptimization task as a stochastic search problem and produces programs either match or outperform the code produced by `gcc -O3`, `icc -O3`. cSTOKE [51] improves STOKE by using the knowledge of input restrictions to generate binaries that ensure correctness only on the restricted inputs. Although superoptimizers perform an iterative optimization process similar in spirit to our search process, they optimize assembly code, which does not have types and thus compatibility issues that HeteroGen has to deal with.

Program Repair. Starting from a faulty program that deviates from its intended behavior, the automated repair process iterates *fault localization*, *candidate repair generation*, and *repair evaluation* [29, 31, 35, 39, 57]. To minimize unfruitful repair attempts, several techniques [9, 23, 30, 47] leverage smart encoding of complex repairs. Some explores the space of repair candidates by leveraging fix patterns learned from correct reference code [36, 66], bug fix histories [31, 34, 39], or human-written patches [29]. Inpired bySYDIT [40] and LASE [41], HeteroGen extracts complex repairs from example patches and encode repairs in terms of parameterized AST edits with dependence relations. However, all repair techniques build on the assumption that the target program can be compiled quickly. On the contrary, HeteroGen uses novel techniques that are specifically designed to address the need of behavior preservation and performance optimization in an environment where a naïve "trial-and-error" approach is prohibitively expensive.

8 CONCLUSION

This paper presents HeteroGen, a C-to-HLS-C transpiler that solves the painful HLS code conversion problem with novel techniques specifically designed to address the need of semantics preservation and performance optimization. HeteroGen produces an HLS-compatible version for nine out of ten real-world heterogeneous applications fully automatically and achieves an overall of 1.63× speedup compared with the input programs.

ACKNOWLEDGMENTS

The participants of this research are supported in part by National Science Foundations via grants CCF-2106404, CNS-2106838, CCF-1764077, CHS-1956322, CCF-1723773, CNS-1763172, CNS-1907352, CNS-2006437, CNS-2007737, CNS-2128653, ONR grant N00014-18-1-2037, ONR grant N00014-16-1-2913, Intel CAPA grant, and Samsung.

REFERENCES

[1] 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/dynamic-memory-allocation-deallocation-is-not-supported-variable/td-p/729976.

[2] 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Error-with-fixed-point-design-in-vivado-HLS/m-p/752508.

[3] 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/dataflow-directive/m-p/595161.

[4] 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Vivado-HLS-loop-unrolling-option-region/m-p/721719.

[5] 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Using-streams-in-objects-does-not-synthesize-in-HLS-2020-1/m-p/1117215.

[6] 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Cannot-find-the-top-function/m-p/810885.

[7] 2021. American Fuzz Loop. http://lcamtuf.coredump.cx/afl/.

[8] Amazon.com. 2021. Amazon EC2 F1 Instances: Run Custom FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1.

[9] Andrea Arcuri. 2008. On the Automation of Fixing Software Bugs. In *Companion of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 1003–1006. https://doi.org/10.1145/1370175.1370223

[10] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 975–985. https://doi.org/10.1145/3338906.3340456

[11] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1011–1023. https://doi.org/10.1145/3377811.3380432

[12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[13] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/3037697.3037754

[14] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. 2018. Best-Effort FPGA Programming: A Few Steps Can Go a Long Way. *arXiv preprint arXiv:1807.01340* (2018).

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

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

[18] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[19] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2018. Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 1 (2018), 35–47. https://doi.org/10.1109/TCAD.2017.2705069

[20] Prabhat Gupta. 2021. Xeon+FPGA Platform for the Data Center. https://research.ece.cmu.edu/calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf.

[21] Mark Harman. 2010. Automated Patching Techniques: The Fix is in: Technical Perspective. *Commun. ACM* 53, 5 (May 2010), 108. https://doi.org/10.1145/1735223.1735248

[22] Intel. 2022. Temporal To Spatial Programming. https://github.com/IntelLabs/t2sp.

[23] Tao Ji, Liqian Chen, Xiaoguang Mao, and Xin Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 197–202. https://doi.org/10.1109/COMPSAC.2016.69

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. https://doi.org/10.1145/3079856.3080246

[25] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. *KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3445814.3446745

[26] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. https://doi.org/10.1109/ISCA.2018.00014

[27] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. 2020. *FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design*. Association for Computing Machinery, New York, NY, USA, 715–731. https://doi.org/10.1145/3373376.3378455

[28] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with Amorphos. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 107–127.

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[30] Ryotaro Kou, Yoshiki Higo, and Shinji Kusumoto. 2016. A Capable Crossover Technique on Automatic Program Repair. In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. 45–50. https://doi.org/10.1109/IWESEP.2016.15

[31] Anil Koyuncu, Kui Liu, Tegawendé Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25 (05 2020). https://doi.org/10.1007/s10664-019-09780-z

[32] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. 2021. *Compiler-Driven FPGA Virtualization with SYNERGY*. Association for Computing Machinery, New York, NY, USA, 818–831. https://doi.org/10.1145/3445814.3446755

[33] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 493–505. https://doi.org/10.1145/3377811.3380340

[34] X. D. Le, D. Lo, and C. Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 213–224. https://doi.org/10.1109/SANER.2016.76

[35] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[36] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[37] LLVM. 2021. https://llvm.org/.

[38] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. *A Hypervisor for Shared-Memory FPGA Platforms*. Association for Computing Machinery, New York, NY, USA, 827–844. https://doi.org/10.1145/3373376.3378482

[39] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Softw. Engg.* 20, 1 (Feb. 2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8

[40] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 329–342. https://doi.org/10.1145/1993498.1993537

[41] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 502–511. https://doi.org/10.1109/ICSE.2013.6606596

[42] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[43] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. *A Compiler Infrastructure for Accelerator Generators.* Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[44] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[45] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. 2012. An empirical study of supplementary bug fixes. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 40–49. https://doi.org/10.1109/MSR.2012.6224298

[46] 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. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22. https://doi.org/10.1109/MM.2015.42

[47] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[48] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015

[49] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[50] Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois. 2019. Module-per-Object: A Human-Driven Methodology for C++-Based High-Level Synthesis Design. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 218–226. https://doi.org/10.1109/FCCM.2019.00037

[51] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[52] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 147–162. https://doi.org/10.1145/2814270.2814278

[53] David B Thomas. 2016. Synthesisable recursion for C++ HLS tools. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 91–98.

[54] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. *Fleet: A Framework for Massively Parallel Streaming on FPGAs.* Association for Computing Machinery, New York, NY, USA, 639–651. https://doi.org/10.1145/3373376.3378495

[55] Valgrind. 2022. https://valgrind.org/.

[56] Chao Wang, Lei Gong, Fahui Jia, and Xuehai Zhou. 2021. An FPGA Based Accelerator for Clustering Algorithms With Custom Instructions. *IEEE Trans. Comput.* 70, 5 (2021), 725–732. https://doi.org/10.1109/TC.2020.2995761

[57] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[58] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 765–777. https://doi.org/10.1145/3377811.3380396

[59] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/bd-p/hls.

[60] Zeping Xue and David B. Thomas. 2015. SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.1109/FPL.2015.7293959

[61] Zeping Xue and David B. Thomas. 2016. SynADT: Dynamic Data Structures in High Level Synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 64–71. https://doi.org/10.1109/FCCM.2016.26

[62] Yue Zha and Jing Li. 2020. *Virtualizing FPGAs in the Cloud.* Association for Computing Machinery, New York, NY, USA, 845–858. https://doi.org/10.1145/3373376.3378491

[63] Yue Zha and Jing Li. 2021. *When Application-Specific ISA Meets FPGAs: A Multi-Layer Virtualization Framework for Heterogeneous Cloud FPGAs.* Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/3445814.3446699

[64] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 722–733.

[65] Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. *HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications.* Association for Computing Machinery, New York, NY, USA, 242–254. https://doi.org/10.1145/3468264.3468610

[66] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 886–896. https://doi.org/10.1145/3180155.3180260

[67] 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. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 269–278. https://doi.org/10.1145/3174243.3174255