HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA

ABSTRACT

Heterogeneous computing with field-programmable gate-arrays (FPGAs) has demonstrated orders of magnitude improvement in computing efficiency for many applications. However, the use of such platforms so far is limited to a small subset of programmers with specialized hardware knowledge. 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.

We propose a new dynamic invariant analysis and automated refactoring technique, called HeteroRefactor. First, HeteroRefactor monitors FPGA-specific dynamic invariants—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 to optimize the accelerator's resource usage and frequency further. Third, to guarantee correctness while leveraging both CPU and FPGA, 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 177 LOC, an expert FPGA programmer would need to write 181 *more* LOC to implement an HLS compatible version, while HeteroRefactor fully automates such edit. The evaluation results on Xilinx FPGA show that HeteroRefactor minimizes BRAM by 85% and increases frequency by 65% for recursive programs; reduces BRAM by 41% through integer bitwidth reduction; and reduces DSP by 50% through floating-point bitwidth reduction.

KEYWORDS

heterogeneous computing, automated refactoring, FPGA, high level synthesis, dynamic analysis

ACM Reference format:
. 2019. HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. In *Proceedings of ICSE '20: Proceedings of the 2020 International Conference on Software Engineering, Seoul, South Korea, May 23–29, 2020 (ICSE '20),* 12 pages.
https://doi.org/00.0000/0000000.0000000

ICSE '20, May 23–29, 2020, Seoul, South Korea

ACM ISBN 000-0-0000-0000-0/00/00…$15.00
https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

In recent years, there has been a growing interest in architectures that incorporate heterogeneity and specialization to improve performance, e.g., [11, 12, 14]. FPGAs are reprogrammable hardware that often exceeds the performance of general-purpose CPUs by several orders of magnitude [10, 49] and offer lower cost across a wide variety of domains [8, 9]. To support the development of such architectures, hardware vendors support CPU+FPGA multi-chip packages (e.g., Intel Xeon [29]) and cloud providers support virtual machines with FPGA accelerators and application development frameworks (e.g., Amazon F1 [3]).

Although FPGAs provide substantial benefits and are commercially available to a broad user base, they are associated with a high development cost [19]. Programming an FPGA is a difficult task; hence, it is limited to a small subset of programmers with specialized knowledge on FPGA architecture details. To address this issue, there has been work on high-level synthesis (HLS) for FPGAs [17]. HLS tools take a kernel written in C/C++ as input and automatically generates an FPGA accelerator. However, to meet the HLS synthesizability requirement, significant code rewriting is needed. For example, developers must manually remove the use of pointers, memory management, and recursion, since such code is not compilable with HLS. To achieve high efficiency, the users must heavily restructure the kernel to supply optimization information manually at the synthesis time. Carefully handcrafted HLS optimizations are non-trivial and out of reach for software engineers who usually program with CPUs [13].

Our observation is that software kernels are often over-engineered in the sense that a program is generalized to handle more inputs than what is necessary for common-case inputs. While this approach has no impact on the program efficiency on a CPU, in an FPGA accelerator, the design efficiency could be impacted considerably by the compiled size that depends on actual ranges of values held by program variables, the actual size of recursive data structures observed at runtime, etc. For example, a programmer may choose a 32-bit integer data type to represent a human age, whose values range from 0 to 120 in most cases. Consider another example, where in 99% of executions, the size of a linked list is bounded by 2k; however, the programmer may manually flatten it to an array with an overly-conservative size of 16k.

We propose a novel combination of dynamic invariant analysis, automated refactoring, and selective offloading approach, called HeteroRefactor to guide FPGA accelerator synthesis. This approach guarantees correctness—*behavior preservation*, as it selectively offloads the computation from CPU to FPGA, only if the invariant is met, but otherwise keeps the computation on CPU. It also does not require having a representative data set for identifying dynamic invariants, as its benefit is to aggressively improve FPGA accelerator efficiency for a common case input without sacrificing correctness. In this approach, a programmer first implements her kernel code in a high-level language like C / C++. Then she executes the kernel code on existing tests or a subset of input data to identify *FPGA-specific dynamic invariants*. Then HeteroRefactor automatically refactors the kernel with pointers into a pointerless, non-recursive program to make it HLS-compatible and to reduce resource usage by lowering bitwidth for integers and floating points, which in turn reduces pre-allocated on-chip elements and increases the frequency at the FPGA level.

We evaluate HeteroRefactor on ten subject programs, including five handwritten recursive programs, three integer-intensive programs from Rosetta benchmark [70], and two floating point-intensive programs from OpenCV [7]. We then generate kernels targeting to a Xilinx Virtex UltraScale+ XCVU9P FPGA on a VCU1525 Reconfigurable Acceleration Platform [67] and achieve the following results:

(1) For recursive programs that are traditionally unsynthesizable, HeteroRefactor refactors pointers and recursion with the accesses to a flattened, finite-size array, making them HLS-compatible. On average, for a recursive program of size 176 LOC, an expert FPGA programmer would need to write 180 *more* LOC to implement an HLS-compatible version, while HeteroRefactor requires no code change. Using a tight bound for a recursive data structure depth, the resulting accelerator is also resource-efficient—an accelerator with a common-case bound of 2k size can achieve 86% decrease in BRAM and 67% increase in frequency compared to the baseline accelerator with an overly conservative size of 16k.
(2) For integers, HeteroRefactor performs transparent optimization and reduces the number of bits by 76%, which leads to 25% reduction in flip flops, 21% reduction in look-up tables, 41% reduction in BRAM, and 52% decrease in DSP.
(3) For floating-point optimization, when an acceptable precision loss is specified as 0.01 at 99.9% confidence level, HeteroRefactor automatically generates an accelerator with 38.2% reduction in flip flops, 28.6% reduction in look-up tables, and 50% reduction in DSP, while providing a probabilistic guarantee that the result differs less than the tolerable loss.

In summary, this work makes the following contributions:

- Traditionally, automated refactoring has been used to improve software maintainability. We adapt and expand automated refactoring to lower the barriers of creating customized circuits using HLS and improving the efficiency of the generated FPGA accelerator.
- While both dynamic invariant analysis and automated refactoring have a rich literature in software engineering, we design a novel combination of dynamic invariant analysis, automated kernel refactoring, and selective offloading, for transparent FPGA synthesis and optimization with correctness guarantee, which is unique to the best of our knowledge.
- We demonstrate the benefits of FPGA-specific dynamic invariant and refactoring in three aspects: (1) conversion of recursive data structures, (2) integer optimization, and (3) floating-point tuning with a probabilistic guarantee.

2 BACKGROUND

This section overviews a developer workflow when using a high-level synthesis (HLS) tool for FPGA and describes the types of manual refactoring a developer must perform to make their kernel synthesizable and efficient on FPGA.

2.1 Overview of FPGA Programming with HLS

Modern FPGAs include millions of look-up tables (LUTs), thousands of embedded block memories (BRAMs), thousands of digital-signal processing blocks (DSPs) [65], and millions of flip-flop registers (FFs). Each k-input LUT can implement any Boolean function up to k inputs. An FPGA needs to be programmed with a specific binary *bitstream* to specify all the LUT, BRAM, DSP, and programmable switch configurations to achieve the desired behavior. Fortunately, HLS has been developed in recent years to aid the translation of algorithmic descriptions (e.g., kernel code in C/C++) to application-specific bitstreams [17, 22, 43]. Specifically, HLS raises the abstraction of hardware development by automatically generating RTL (Register-Transfer Level) descriptions from algorithms. Generation of FPGA-specific bitstream consists of a *frontend* responsible for C simulation and a *backend* responsible for hardware synthesis. In the frontend, after analysis of C/C++ code, HLS *schedules* each operation from the source code to certain time slots (clock cycles). Next, it allocates *resources*, i.e., the number and type of hardware units used for implementing functionality, like LUTs, FFs, BRAMs, DSPs, etc. Finally, the *binding* stage maps all operations to the allocated hardware units. This frontend process generates an RTL, which is sent to a backend to perform logic synthesis, placement, and routing to generate FPGA bitstreams. Software simulation is fast; however, hardware synthesis can take anywhere from *a few hours to a couple of days*, depending on the complexity of the algorithm.

Therefore, such long hardware synthesis time contributes to the cost of a developer rewriting their kernel for optimized resource allocation, frequency, and power utilization. The current FPGA design flow also motivates HeteroRefactor's approach to invest time in apriori dynamic analysis as opposed to just-in-time compilation techniques to optimize FPGA design, as frequent iterations of hardware synthesis are prohibitively expensive.

2.2 Refactoring for High-Level Synthesis

HLS tools aim to narrow the gap between the software program and its hardware implementation. While HLS tools take kernel code in C or C++, a developer must perform a substantial amount of manual refactoring to make it *synthesizable* and *efficient* on an FPGA chip. Such refactoring is error-prone and time-consuming since certain language constructs for readability and expressiveness in C/C++ are not allowed in HLS [19]. A developer must have interdisciplinary expert knowledge in both hardware and software and know obscure platform-dependent details [13]. Below, we categorize manual refactorings for HLS into two kinds: (1) *synthesizability* and (2) *efficiency optimization*. In this paper, we focus on improving the Vivado HLS tool from Xilinx [17, 66], which is the most widely used FPGA HLS in the community, although our techniques can be easily generalized to other HLS tools, such as Intel HLS Compiler, Catapult HLS from Mentor, and CyberWorkBench from NEC.

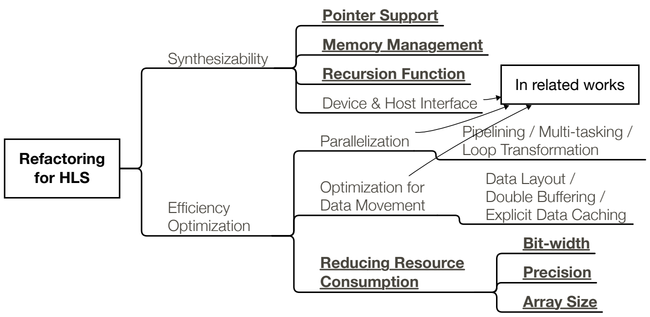

Figure 1: Overview of Refactoring for High-Level Synthesis

2.2.1 *Synthesizability.*

Pointer support. To transform kernel code into its equivalent HLS synthesizable version, a developer must manually eliminate pointer declarations and usages; there are only two types of pointers that are natively supported in HLS—pointers to hardware interfaces such as device memory or pointers to variables. Pointer casting is limited to primitive data types. Arrays of pointers or recursive data structures are strictly forbidden in Vivado HLS.

Memory management and recursion. Because Vivado HLS has no capability of *memory management*, function calls to memory allocation such as `malloc` cannot be synthesized. Thus, developers must create an overly conservative, large-sized static array in advance and manage data elements manually. Similarly, Vivado HLS cannot synthesize recursion. Thus, developers must manually convert recursions into iterations or create a large stack to store program states and manage function calls manually.

Device and host interface. Vivado HLS requires a strict description of parameters of the top-level function that acts as the *device and host interface*. The function is called from the host and is offloaded into FPGA. The type of a function parameter can be either a scalar or pointer to the device memory with a data size in the power of 2 bytes, and a developer must write specific pragmas—for example, `#pragma HLS interface m_axi port=input` to use AXI4 interconnect interface for passing the parameter named `input` to the FPGA design.

2.2.2 *Efficiency Optimization.*

Parallelization. Reprogrammable hardware provides an inherent potential to implement *parallelization*. Such parallelization can be done through pipelining of different computation stages and by duplicating processing elements or data paths to achieve an effect similar to multi-threading. To guide such parallelization, a developer must manually write HLS pragmas such as `#pragma HLS pipeline` and `#pragma HLS unroll` for suitable loops or must expose parallelization opportunities through polyhedral model-based loop transformations [5, 6].

Optimization of data movement. We can access device memory more efficiently by packing bits into the width of DRAM access of 512 bits. To overlap communication with computation, a developer must explicitly implement a double buffering technique [13]. To cache data, developers need to explicitly store them on chip through data tiling or batch processing of tasks [48, 55].

Reducing resource consumption. Provisioning more processing elements or a larger cache will require using more on-chip

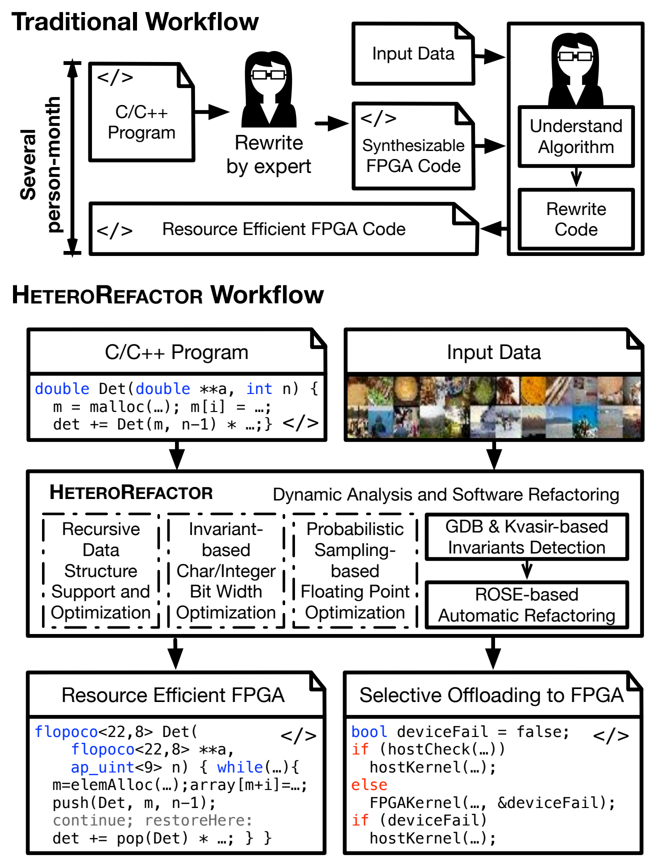

Figure 2: Approach Overview of HeteroRefactor

resources, limiting the potential of parallelization and data movement optimizations by duplicating processing elements or adding cache. A higher resource utilization ratio can lower the maximum operating frequency and consume more power, thus, it degrades the performance and efficiency. Besides, a resource-efficient design is economical as it can to be implemented on a smaller FPGA chip. Traditionally, developers allocate integers and floating-point variables with a fixed size bitwidth large enough for all possible input values, or create a static array for the largest-possible size. Such a practice may cause wasting on-chip resources. In particular, in modern applications such as big data analytics and ML applications where on-chip resource usage is input-dependent, FPGA resource optimization becomes increasingly difficult.

Figure 1 highlights our new contributions, highlighted with **underline** and **bold**, relative to the prior HLS studies. There exists many automated approaches for generating device and host interfaces [16, 69], exploring parallelization opportunities [18, 39, 69], and optimizing data movement [16, 18, 39, 47, 48]. But general methods for reducing resource consumption, pointer support, memory management and recursion support remain as *open research questions* and no automated kernel refactoring exists yet. HeteroRefactor addresses three important scopes of such refactoring transformations: (1) converting a program with pointers and recursion to a pointerless and non-recursive program by rewriting memory

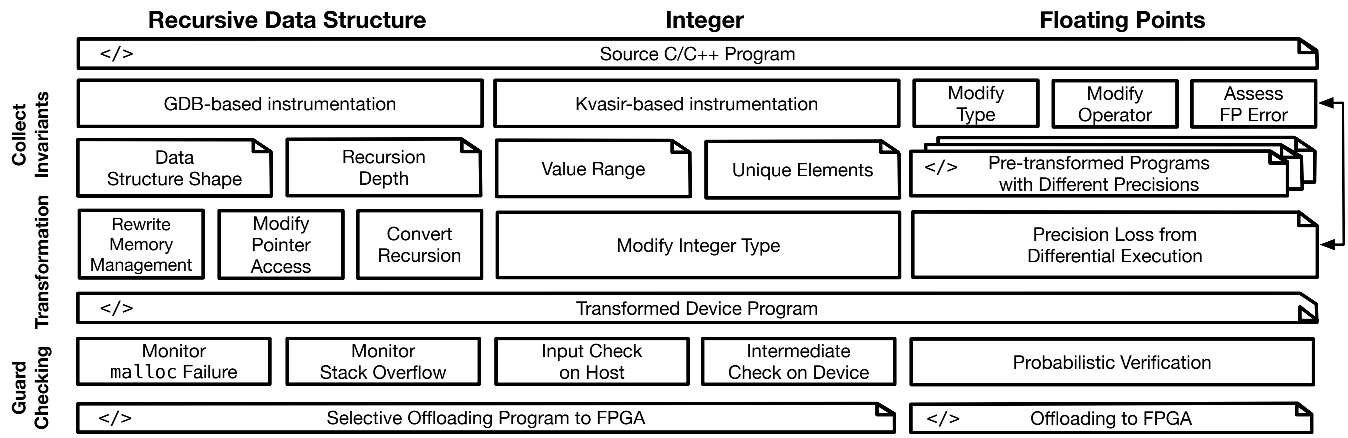

Figure 3: HeteroRefactor incorporates three techniques—dynamic invariant detection, kernel refactoring, and selective offloading with guard checking. Its profiling concerns three aspects: (1) the length of recursive data structures, (2) required integer bitwidth, and (3) required floating-point bitwidth to meet a specified precision loss.

management and function calls, (2) reducing on-chip resource consumption of integer bitwidth, and (3) reducing on-chip resource consumption by tuning floating-point precision.

3 APPROACH

HeteroRefactor, as shown in Figure 2, is a novel end-to-end solution that combines dynamic invariant analysis, automated refactoring, and selective offloading for FPGA. It addresses three kinds of HLS refactorings: rewriting a recursive data structure to an array of finite size (Section 3.1); reducing integer bitwidth (Section 3.2); and tuning variable width floating-point operations (Section 3.3). Figure 3 details the three components of HeteroRefactor that work in concert: (A) instrumentation for FPGA-specific dynamic invariant analysis such as the required bitwidth of integer and floating-point variables and the size of data structures and stacks, (B) source-to-source transformation using the knowledge of dynamic invariants, and (C) selective offloading that checks the guard condition when offloading from CPU to FPGA. The first two kinds of refactorings follow similar implementation for selective offloading using a guard condition check, described in Section 3.4. For floating-point operations, our dynamic analysis provides a probabilistic guarantee that the precision loss will be within a user-specified bound.

3.1 Recursive Data Structure

Many applications use recursive data structures built on `malloc`, `free`, and recursive function calls. As mentioned in Section 2.2, HLS tools have strict restrictions on the types of pointers allowed and do not support memory allocation and recursion. For example, Vivaldo HLS throws the following error for Figure 4: *an unsynthesizable type '[10 x %struct.Node.0.1.2]**. This severely limits the type of programs that can be automatically ported for heterogeneous computing. Expert FPGA developers manually rewrite the recursive data structure into a flattened array to be HLS-compliant; however, as they may not know the common maximum size required for the application, they often over-provision and declare an unnecessarily large size. They also have to manually convert recursion into loop iterations and over-provision the stack required for keeping track of program state involved in recursive calls.

HeteroRefactor uses GDB-based instrumentation to identify the size of recursive data structures and the corresponding stack depth and performs source-to-source transformation based on the size.

3.1.1 *GDB-based Instrumentation.*

We use `GDB` debugger [59] to develop a lightweight memory graph tracing capability. HeteroRefactor uses Python APIs of `GDB` to track the underlying memory usage of all variables of the recursive data structure under focus and the corresponding stack depth of recursive function calls.

HeteroRefactor automatically steps through the program and monitors the size of the data structure at each instrumentation point, for example, at the end of insertion. HeteroRefactor then determines the percentiles of the sizes. In Figure 4, given the instrumentation line number, HeteroRefactor sets a breakpoint at line 14 and records the size of `root`. To monitor the depth of recursion, we set two breakpoints, `BP1` at the function entry point and `BP2` at the function exit point of each recursive function. As an example, in Figure 4, `BP1` is set at line 6 and `BP2` is set after line 9. HeteroRefactor then maintains a `GDB` variable `stack_size`, which is incremented every time the program reaches `BP1` and is decremented when it reaches `BP2`. The highest value attained by `stack_size` during execution is reported and used as the bound for a flattened array and the corresponding stack.

3.1.2 *Refactoring.*

HeteroRefactor is implemented based on a source-to-source compiler framework, ROSE [50], to rewrite recursive data structures. It takes C/C++ kernel code and the array sizes and recursion depths found via dynamic analysis, and outputs an HLS-compatible version with implementation of on-chip memory allocation, removes all pointers except for those with native HLS support (to be explained further under Rule 2), and rewrites recursive functions. The transformation is semantics-preserving and consists of the following transformation rules:

Rule 1: Rewrite Memory Management. To replace calls to `malloc` and `free`, for each data type, we pre-allocate an array whose size is guided by instrumentation (line 3 in Figure 5). The per-element

```
1  struct Node { Node *left, *right; int val; };
2  void init(Node **root) {
3    *root = (Node *)malloc(sizeof(Node)); }
4  void insert(Node **root, int n, int *arr);
5  void traverse(Node *curr) {
6    if (curr == NULL) return;
7    visit(curr->val);
8    traverse(curr->left);
9    traverse(curr->right); }
10 void top(int n, int *output_if) {
11 #pragma HLS interface m_axi port=output_if
12   Node *root; init(&root); // ...
13   int values[3] = {5, 4, 3};
14   insert(&root, 3, values);
15   int *curr = output_if; traverse(root); // ...
16   free(root); }
```

Figure 4: Binary tree using pointers and memory allocation

```
1  bool guard_error = false;
2  struct Node { Node_ptr left, right; int val; };
3  struct Node Node_arr[NODE_SIZE];
4  typedef unsigned int Node_ptr;
5  Node_ptr Node_malloc(size_t size);
6  void Node_free(Node_ptr); // buddy allocation
7  void init(Node_ptr *root) {
8    *root = (Node_ptr)Node_malloc(sizeof(Node));
9    if (!root) guard_error = true; }
10 void insert(Node_ptr *root, int n, int *arr);
11 void traverse(Node_ptr curr) {
12   stack<context> s(TRAVERSE_STACK_SIZE, {curr:curr,loc:0});
13   while (!s.empty()) { context c = s.pop(); goto L{c.loc};
14   L0:if (c.curr == NULL) continue;
15     visit(Node_arr[c.curr-1]._data.val);
16     if (s.full()) { guard_error=true; return; }
17     c.loc = 1; s.push(c); s.push({curr: Node_arr[
18       c.curr-1]._data.left, loc: 0}); continue; // traverse(left)
19   L1:// traverse(right) ...
20   L2:; } }
21 void top(int n, int *output_if, bool *fail) {
22   #pragma HLS interface m_axi port=output_if
23   Node_ptr root; init(&root); // ...
24   int values[3] = {5, 4, 3};
25   insert(&root, 3, values);
26   int *curr=output_if; traverse(root); // ...
27   Node_free(root); *fail = guard_error; }
```

Figure 5: Refactored binary tree (schematic)

allocation strategy with an array is based on two reasons—HLS only supports pointer casting on primitive data types, and it can optimize array accesses if the size of one element is known. For each node allocation and de-allocation, we implement a buddy memory system [46] and allocate from the array. The buddy memory system requires less overhead and has little external fragmentation [64], making it suitable for FPGA design. We identify all calls to `malloc` and `free`, the requested types and element counts, and transform them into calls to our library function `Node_malloc` (line 8 in Figure 5) which returns an available index from the array. Section 4.1 details performance benefits in terms of increased frequency and reduced resource utilization using an array size guided by dynamic analysis rather than declaring an overly conservative size.

Rule 2: Modify Pointer Access to Array Access. There are only two types of pointers *natively supported* in HLS, and we do not need to convert them into array access. One is a pointer of interfaces, which we can identify by looking up pragmas in the code (line 22 in Figure 5). Second is a pointer to variables, which can be detected by finding all address-of operators or array references in the code. Before modifying pointer access to array access, we identify these natively supported pointers using a breadth-first search on the data flow graph and exclude them from our transformation.

We transform the pointers to an unsigned integer type that takes value less than the size of the pre-allocated array from dynamic

```
int weakClassifier(int stddev, int coord[12], int haarC, int w_id);
int cascadeClassifier(int SUM1_data[IMG_HEIGHT][IMG_WIDTH],
     int SQSUM1_data[IMG_HEIGHT][IMG_WIDTH], MyPoint pt) { // ...
  int stddev = int_sqrt(stddev); // ..
}
```

Figure 6: Original code from `Face Detection`

```
bool guard_error = false;
void guard_check(ap_int<65> value, int size, int sign) {
  #pragma HLS inline off
  if (sign==1) { if (value<0) {
    if (value < -(1LL<<(size-1)) guard_error = true;
  } else { /*...*/ } } else { /*...*/ } }
int weakClassifier(ap_uint<9> stddev, ap_uint<23> coord[12],
       ap_uint<7> haarC, ap_uint<8> w_id);
int cascadeClassifier(ap_uint<18> SUM1_data[IMG_HEIGHT][IMG_WIDTH],
    ap_uint<18> SQSUM1_data[IMG_HEIGHT][IMG_WIDTH], MyPoint pt) { //...
  ap_uint<18> stddev = int_sqrt(stddev);
  guard_check(ap_int<65>(int_sqrt(stddev)),18,0); // ...
}
```

Figure 7: Refactored code for `Face Detection`

analysis. This integer represents the offset of the pointed element in the pre-allocated array. There are three locations where this type of transformation is applied: (1) variable declarations (line 23 Figure 5), typecasting (line 8 Figure 5), and function parameters (line 10 Figure 5) and the return value in both declarations and the definition. We perform a breadth-first search on the data flow graph to propagate the type changes. Since we use an array offset to reference allocated elements, we need to change all pointer dereferences into array accesses with the relative index. We transform indirection operator (`*ptr`) and structure dereference operators (`ptr->, ptr->*`) into array accesses with pointer integer as the array index. Similarly, the subscript operators (`ptr[]`) are transformed into array accesses with the pointer integer added with the given offset as the array index. For example, we modify pointer access (line 7 in Figure 4) to array access (line 15 in Figure 5).

Rule 3: Convert Recursion to Iteration. To transform recursive functions into non-recursive ones, we create a stack (line 12 Figure 5) for each function with all local variables in its nodes. The depth of the allocated stack is determined through the dynamic analysis step. All references to local variables are transformed into references to elements on the top of the stack (line 14 Figure 5). To simulate the saved context of the program counter and return value in a CPU call stack, we reserve two member variables in our stack to store the location indicating which line of code we need to restore to, and the return value of the called function.

With a stack, we can implement function calls like in CPU. Entering a function pushes the current context and new parameters to the top of the stack (line 17 Figure 5), then continue to the first line of the function (line 18 Figure 5). A function `return` writes the return value to the stack, pops the top item from the stack, and returns to the saved context (lines 13-14 Figure 5).

3.2 Integer

3.2.1 Kvasir-based Instrumentation. Daikon is a dynamic invariant detection tool [21] that reports likely program invariants during a program's execution. It consists of two parts: (a) a language-specific front-end and (b) a language-independent inference engine. A front end instruments the program and extracts the program

```
float l2norm(float query[], float data[], int dim) {
  float dist = 0.0;
  for (int j = 0; j < dim; j++)
    dist += ((query[j] - data[j]) * (query[j] - data[j]));
  return sqrt(dist); }
```

Figure 8: `l2norm` from KNN

```
using namespace thls; typedef policy_flopoco<16,5>::value_t LOWBIT;
float low_l2norm(float query[], float data[], int dim) {
  LOWBIT dist = 0.0;
  for (int j = 0; j < dim; j++) {
    LOWBIT fp_query_j = to<LOWBIT, policy>(query[j]);
    LOWBIT fp_data_j = to<LOWBIT, policy>(data[j]);
    LOWBIT fp_neg_1 = neg(fp_data);
    dist += (fp_query + fp_neg_1) * (fp_query + fp_neg_1); }
  return sqrt(to<float>(dist)); }
int main() {
  for (...) { // ...
    float highValue = l2norm(args[]);
    float lowValue = low_l2norm(args[]);
    float currentError = highValue - lowValue;
    if (fabs(currentError) > acceptaErrorz) Failed++; else Passed++; }
  if (double(Passed) / Samples > requiredProbability) {
    /* Passed verification */ } else { /* Failed verification */ } }
```

Figure 9: Refactored `l2norm` that performs differential execution and probabilistic verification (schematic)

state information by running the program. FPGA kernels are programmed in C/C++ for HLS; hence, we use Kvasir [21], a C/C++ front-end for Daikon, to instrument the target program's binary.

3.2.2 FPGA-Specific Invariants. Daikon is often used for general program comprehension and testing, and therefore it outputs invariants such as an array size or binary comparison, e.g., `i>0`, `i<0`, `size(array)=0`, `size(array)>0`. However, such general invariants must be adapted for the purpose of FPGA synthesis. For example, reducing the variable bitwidth leads to resource reduction in FPGA [38] directly.

Therefore, to optimize FPGA synthesis, we design three types of FPGA-specific invariants: (1) the minimum and maximum value of a variable based on a range analysis, (2) the number and type of unique elements in an array, and (3) the size of an array. For example, first consider Figure 6. A programmer may over-engineer and use a 32-bit integer by default, which is a higher bitwidth than what is actually necessary. While the instruction set architecture (ISA) for CPU defines integer arithmetics at 32 bits by default, in FPGA, individual bitwidths could be programmed.

3.2.3 Refactoring. Rule Modify Variable Type. To automatically convert an integer to an arbitrary precision integer, we leverage `ap_uint<k>` or `ap_int<k>`, which defines an arbitrary precision integer of `k` bits. As an example, the input `haar_counter` to method `weakClassifier` in Figure 6 is declared as a 32-bit `integer` by the programmer. However, suppose that HeteroRefactor finds that it has a min value of 0 and a max value of 83—it then only needs 7 bits instead of 32 bits. HeteroRefactor parses the program's AST using ROSE [50], identifies the variable declaration node for `stddev`, `coord`, and `haar_counter` with `w_id` type, and then modifies the corresponding type as shown in Figure 7.

3.3 Floating Point

Unlike the reduction of integer bitwidth in the previous section, reducing the required bitwidth for floating-point (FP) variables can lead to FP precision loss. Estimating the error caused by lowering a FP bitwidth can be done reliably only through differential execution, because existing static analysis tends to over-approximate FP errors. Therefore, we design a new probabilistic, differential execution-based FP tuning approach, which consists of four steps: (1) source-to-source transformation for generating program variants with different biwidths, (2) estimation of the required number of input data samples based on Hoeffding's inequality [31], (3) test generation and differential execution, and (4) probabilistic verification for FP errors.

Prior work on reducing FP precision in CPU [53, 54] used dynamic analysis; however, since they use a golden test set, they do not provide any guarantee on running the reduced precision program on unseen inputs. The key insight behind HeteroRefactor's probabilistic verification approach is that we can draw program input samples to empirically assess whether the relative error between a low precision program and a high precision program is within a given acceptable precision loss e with probability α. Given a program with high-precision FP operations, hp, we construct a lower precision copy of the program, lp, by changing the corresponding type of all FP parameters, local variables, and constants. For each input $i \in I$, we compute the actual error between the high and the low bitwidth variant, $hp(i) - lp(i)$. We then check whether this loss is less than the acceptable precision loss e. This predicate $hp(i) - lp(i) < e$ now forms a binomial distribution B, which checks whether the actual FP error is within the acceptable loss for the given input distribution.

HeteroRefactor takes as inputs: (1) a program p, (2) an acceptable loss (error) e, (3) a set of sampled inputs I or a statistical distribution, (4) a required probability α, and (5) deviation ϵ. We use Hoeffding's inequality [31] to compute the minimum number of samples required to satisfy a given probability α and deviation ϵ. Equation 1 shows that the empirical expectation $E[i]$ of the binomial distribution B deviates from its true expectation p by ϵ with probability less than 1-α. Similar to Sampson et al.'s probablistic assertion [56], we use Hoeffding's inequality since it provides a conservative, general bound for expectations of *any arbitrary distribution* and relies only on probability and deviation. Therefore, it is suitable for our situation where we have no prior knowledge about the FP loss distribution, incurred by reducing the bitwidth. Equation 2 calculates the minimum number of samples required to verify whether the error is within the acceptable loss e.

$$P[E[i] - p \geq \epsilon] \leq e^{-2n\epsilon^2} \quad (1)$$

$$n \geq -ln(1-\alpha)/(2\epsilon^2) \quad (2)$$

For example, when a user wants the actual FP error between the high precision variant (Figure 8) and the low precision variant (Figure 9) to be less than 10^{-4} with 95% probability and 0.03 deviation, the minimum number of samples required is 2049.

During differential testing with respect to input I, if the proportion of passing samples to $|I|$ is greater than α, we probabilistically guarantee that it is safe to lower the FP precision to the given lower bitwidth. The following transformation rules are applied to identify a lower precision configuration for FP variables.

Rule 1. Duplicate Method and Modify Type. To create multiple copies of method `l2norm` in Figure 8, HeteroRefactor traverses its AST and redefines the type of variable `query`, `data`, and `dim` originally declared as `float` using `thls::fp_flopoco<E, F>`, whose library is based on Thomas' work on *templatized soft floating-point type for HLS* [62]. `E` is the number of exponent bits and `F` is the number of fractional bits (excluding 1 implicit bit). For example, `thls::fp_flopoco<8,23>` is 32 bit `float` type, and `thls::fp_flopoco<5,16>` uses 22 bits in total (5 for exponent, 16 for fraction and 1 for implicit bit) instead.

Rule 2: Modify Arithmetic Operators. While addition, multiplication, and division operators are implemented by `thls::fp_flopoco<E,F>`, subtraction is not supported [62]. Hence, we convert subtraction in `l2norm` (line 4 in Figure 8) to corresponding `neg` and `add`, i.e., $subtract(a, b) = add(a, neg(b))$ using a variable `fp_neg_1` to store the intermediate result (lines 7-8 in Figure 9).

Rule 3: Assess FP Error for Differential Execution. We define a skeleton method that computes the relative error and probabilistically verifies if the error is within the user given acceptable loss (lines 11-17 Figure 9). This involves adding code to invoke the original and generated low precision variant of the function.

3.4 Selective Offloading with Guard Check

To selectively offload the computation that fits the reduced size, we insert guard conditions in the host (function sending data from CPU to FPGA) and the kernel (algorithm) to be mapped to FPGA.

For recursive programs, as illustrated at line 9 and line 16 in Figure 5, we insert a guard condition at the `Node_malloc`. The condition sets a global variable `guard_error` to true, if the array is full and more allocation is required. Similarly, the global variable is set to true, if the stack size grows beyond the reduced size. For integer-intensive programs, as shown at line 11 in Figure 7, we add a guard condition in the kernel and host program. We guard the use of each input, output, and intermediate value in the kernel to proactively prevent overflow (lines 4-6 in Figure 7). For this, we first identify all assignment statements to the reduced bitwidth variables, and if the right-hand side contains binary operations, we insert a guard function.

4 EVALUATION

Our evaluation seeks to answer the following research questions:

RQ1 Does HeteroRefactor effectively enlarge the scope of HLS synthesizability for recursive data structures?
RQ2 How much manual effort can HeteroRefactor save by automatically creating an HLS-compatible program?
RQ3 How much resource reduction does HeteroRefactor provide for recursive data structures, integer optimization, and floating-point optimization?

Benchmarks. We choose ten programs, listed in Table 1 as benchmarks for our main evaluation. For recursive data structures, we use the following five kernels: (R1) `Aho-Corasick` [2] is a string pattern searching algorithm that uses breadth-first search with a dynamic queue, a recursive Trie tree [20] and a finite state machine. (R2) `DFS` is depth-first search implemented with recursion. (R3) `Linked List` is insertion, removal, and sorting on a linked list. (R4) `Merge Sort` is performed on a linked list. (R5) `Strassen's` [33] is a recursive algorithm for matrix multiplication. For integer optimization, we use `face detection` and `3D rendering` from Rosetta [58, 70] (I6 and I7). We also write `bubble sort` (I8). For FP bitwidth reduction, we modify two programs—`KNN` (F9) and `RGB2YUV` (F10) from OpenCV examples [7].

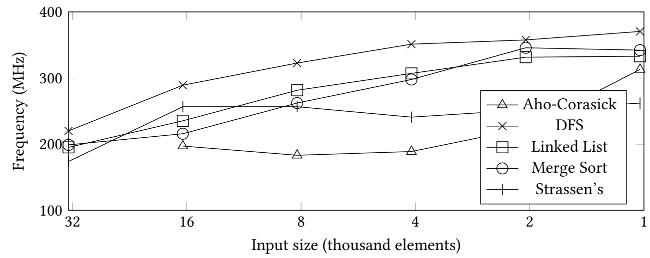

Figure 10: Operating frequency for different input sizes for data structure

Though the code size of subject programs looks small, these programs are rather sizable compared against well-known FPGA HLS benchmarks [30, 52]. Similar to creating a new instruction type in the CPU instruction set architecture, the role of FPGA is to create high performance, custom operators at the hardware circuit level. In fact, in a usual FPGA development workflow, developers instrument software on CPU, find out its hotspot corresponding to tens of lines of code, and extract it as a separate kernel for FPGA synthesis. Therefore, our work cannot be judged under the same scalability standard used for pure software refactoring (e.g., handling GitHub projects with millions of lines of code).

Experimental Environment. All experiments are conducted on a machine with Intel(R) Core(TM) i7-8750H 2.20GHz CPU and 16 GB of RAM running Ubuntu 16.04. The dynamic invariant analysis is based on instrumentation using Daikon version 5.7.2 with Kvasir as front-end and GDB version 8.2.1. The automated refactoring is implemented based on ROSE compiler framework [50] version 0.9.11.0. The refactored programs are then synthesized to RTL to estimate the resource utilization by Vivado Design Suite 2018.03. The tool generated kernels are targeted to a Xilinx Virtex UltraScale+ XCVU9P FPGA on a VCU1525 Reconfigurable Acceleration Platform.

4.1 Recursive Data Structure

To answer `RQ1`, we assess how many of the recursive data structure programs are now synthesizable using HeteroRefactor that fail compilation with Vivado HLS. For `RQ2`, we measure the manual porting effort as LOC and characters in the code. For `RQ3`, we assess the reduction in resource utilization and increase in frequency of the resulting FPGA design code compared to the FPGA design based on a manually written kernel with a conservative size.

Table 2 shows how many lines of code (Manual LOC) and characters (Manual Chars) we need to write in total, if we manually refactor a synthesizable version in Vivado HLS. These manual versions have only a naïve allocator that returns the first unallocated element in the array. If we add a buddy memory system to the manually refactored code to achieve the same functionality as in HeteroRefactor, about 100 additional lines of code are required, and thus manual refactoring effort would be even greater.

To evaluate reduction in resource utilization, we instrument the programs using randomly generated input data where 99% of them have typical sizes of 1k, 2k, 4k or 8k, and others have 16k elements.

Table 1: Resource Utilization for HeteroRefactor

ID / Program		#LUT	#FF	BRAM	DSP
R1 / Aho-Corasick	Orig	Not Synthesizable			
	HR-16K	7541	7651	2298	9
	HR-8K	7614	6247	1152	9
	HR-2K	6959	6152	347	9
R2 / DFS	Orig	Not Synthesizable			
	HR-16K	2988	2962	523	0
	HR-8K	2720	2936	264	0
	HR-2K	2637	2916	69	0
R3 / Linked List	Orig	Not Synthesizable			
	HR-16K	4074	4160	645	0
	HR-8K	3960	4139	318	0
	HR-2K	3759	3983	83	0
R4 / Merge Sort	Orig	Not Synthesizable			
	HR-16K	2949	3006	723	0
	HR-8K	2842	2992	367	0
	HR-2K	2662	2986	105	0
R5 / Strassen's	Orig	Not Synthesizable			
	HR-16K	19972	15095	212	12
	HR-8K	19972	15095	212	12
	HR-2K	19232	14907	59	12
I6 / Face Detection	Orig	11325	5784	49	39
	Manual	10158	4800	49	37
	HR	10298	4770	47	28
I7 / 3D Rendering	Orig	3828	2033	123	36
	Manual	2239	1357	67	12
	HR	1907	878	39	9
I8 / Bubble Sort	Orig	313	125	2	0
	Manual	306	125	1	0
	HR	302	125	1	0
F9 / KNN	Orig	88843	18591	30	32
	α	10^{-2}			
	0.95	80163	15257	30	16
	0.99	82228	15626	30	16
	0.999	82228	15626	30	16
	α	10^{-4}			
	0.95	88952	17102	30	32
	0.99	88952	17102	30	32
	0.999	90441	17855	30	32
	α	10^{-6}			
	0.95	88843	18591	30	32
	0.99	88843	18591	30	32
	0.999	88843	18591	30	32
F10 / RGB2YUV	Orig	398444	73437	30	288
	α	10^{-4}			
	0.95	243516	28379	30	144
	0.99	250044	28827	30	144
	0.999	250044	28827	30	144
	α	10^{-5}			
	0.95	304956	49468	30	144
	0.99	304956	49468	30	144
	0.999	311532	49964	30	144
	α	10^{-6}			
	0.95	372236	66381	30	288
	0.99	398444	73437	30	288
	0.999	398444	73437	30	288

Table 2: Recursive Data Structure Kernels, No Extra Code with HeteroRefactor v.s. Effort for Manual Refactoring

Program	Orig. LOC	Manual LOC	Δ LOC	Orig. Chars	Manual Chars	Δ Chars
R1/A.-C.	188	369	49%	5552	16862	67%
R2/DFS	87	196	56%	2177	7089	69%
R3/L. List	126	206	39%	2976	7404	60%
R4/M. Sort	157	282	44%	3620	10157	64%
R5/Strassen's	326	735	56%	9637	40971	76%
Geomean			48%			67%

The profiled information is then passed to HeteroRefactor, which automatically generates Vivado HLS-compilable variants of the original program. As mentioned in Section 3.1, FPGA programmers manually transform pointer to non-pointer programs with an overly conservative estimate for the size of the data structure. To compare traditional code rewriting to HeteroRefactor, we manually convert the programs in Table 2 with a conservative size of `16k`.

Rows R1-R5 in Table 1 summarize reduction in pre-allocated array size and resource utilization for each of these variants. Manual shows resource usage numbers for refactored program with a conservative size of `16k`, and `HR-8k` and `HR-2k` show resource usage of HeteroRefactor with 8k and 2k typical data size. If the typical input data size is 2k, on average, there is 8% decrease in LUT, 20% decrease in FF and 85% decrease in BRAM usage compared to the refactored program with conservative pre-allocated array sizes. The decrease in consumption of LUTs and FFs is small because the pre-allocated array is large and not partitioned, while the decrease in BRAM is significant because Vivado HLS stores most of the large array in BRAM.

We implement FPGA accelerator on-board with a target frequency of 300 MHz. Figure 10 reports the maximum operating frequency after placement and routing by Xilinx Vivado for each typical input data size. The frequency is calculated statically by using the worst negative slack (WNS) in the report file: $F_{max} = 1/(1/300\text{MHz} + WNS)$. If we are overly conservative on the number of elements and manually refactor the program with an estimation of `32k`, two kernels `Merge Sort` and `Strassen's` will even fail to generate bitstream as they require too many resources, therefore no frequency numbers are reported. In other words, such computation cannot even map to a custom circuit on the correct FPGA. If the input recursive data structure size is 2k, on average, there is 65% increase in frequency compared to the code with a conservative size of `32k`. The frequency improvement comes from reducing communication time among distributed storage resources. When the array is large, the required storage is more distributed, and thus the routing paths are longer, which harms timing.

Summary 1

By identifying an empirical bound for the recursive data structure size, HeteroRefactor makes programs HLS-synthesizable. The accelerators optimized for common-case inputs are 85% more memory-efficient with 65% higher frequency than hand-written code with a conservative size.

Table 3: FPGA’s specific Invariants for Integer Opmization

Program	Variable	FPGA-specific Invariants			
		Min	Max	Unique	Size
I6/Face D.	Weights Array 1	8192	12288	2	2913
I6/Face D.	Stddev Variable	305	369	N/A	N/A
I6/Face D.	Coord	0	6746969	21	12
I7/3D R.	Triangle 3D (x0)	38	255	49	100
I8/B.Sort	Input Array	0	10	11	400

4.2 Integer

We assess the hypothesis that reducing bitwidth based on dynamic invariants leads to reduction in resource utilization for Integers. For this, we measure the resource utilization for each program using Vivado HLS 2018.03 targeting a Xilinx XCVU9P FPGA.

For integer bitwidth reduction, HeteroRefactor takes as input (1) the kernel under analysis and (2) input data. We generate invariants (e.g., Table 3) to create a bitwidth optimized program (e.g., Figure 7). Table 3 reports the *FPGA-specific dynamic invariants* for integer variables. In terms of input data, we either generate synthetic data of a fixed size or use an existing test set. For `Face Detection` we use hex images generated from [58] and resize them to dimension 16x16. The program uses pre-trained weights which are declared as an integer array. HeteroRefactor identifies that one of the weight arrays requires only unsigned 14 bits based on the max and min value and has only two unique values. For `3D Rendering`, we use the test input available in the benchmark [70] and split it into subsets of 100 for each instrumented run. HeteroRefactor identifies that the input model has a range of `(38,150)` and size of 100. For `Bubble Sort`, we generate 400 integers based on Chi Square distribution [40]. The invariants identified by HeteroRefactor reconcile with the distribution parameters and fixed size of the input set.

Rows I6-I8 in Table 1 summarize the bitwidth reduction and resource utilization. For each resource type we report the numbers for (1) original unoptimized program in row `Orig`, (2) manually optimized program by an expert FPGA programmer in row `Manual`, and (3) optimized program by HeteroRefactor in row `Hetero`. On average, HeteroRefactor leads to 25% reduction in FF, 21% reduction in LUT, 41% reduction in BRAM, and 52% decrease in DSP compared to unoptimized program. Compared to carefully hand-crafted programs by experts, HeteroRefactor leads to 12% reduction in FF, 5% reduction in LUTs, 15% reduction in BRAM, and 16% decrease in DSP. Due to the area reduction, more processing elements can be synthesized in one single chip.

We then implement these FPGA accelerators on-board with a target frequency of 300 MHz. All of the refactored programs can meet this target; however, the original version of `3D Rendering` fails the timing constrains and can only work with a final frequency of 240.6 MHz. This validates that the frequency improvement can be achieved by HeteroRefactor.

Summary 2

HeteroRefactor reduces the manual refactoring effort by automatically finding the optimized bit width for integers. It reduces 25% FF, 21% LUTs, 41% BRAM, and 52% DSP in resource utilization, which are better than hand-optimized kernels written by experts.

Table 4: Probabilistic Floating Point Verification

Program	α	10^{-2}			10^{-4}			10^{-6}		
		8	16	HR	8	16	HR	8	16	HR
F9/KNN	0.95	N	N	24	N	N	29	N	N	32
	0.99	N	N	25	N	N	29	N	N	32
	0.999	N	N	25	N	N	30	N	N	32
	α	10^{-4}			10^{-5}			10^{-6}		
F10/R2Y	0.95	N	N	21	N	N	25	N	N	30
	0.99	N	N	22	N	N	25	N	N	32
	0.999	N	N	22	N	N	26	N	N	32

4.3 Floating Point

We evaluate the effectiveness of HeteroRefactor in providing probabilistic guarantee while lowering bitwidth, and reducing the resource utilization compared to original programs.

We begin with the given float (32-bit) precision and generate program variants with a reduced operand bitwidth. Reducing mantissa bits leads to precision loss, whereas reducing exponent leads to a smaller dynamic range. Hence, in our experiments, we incrementally reduce mantissa and verify if the loss is within user given e. As described in Section 3.3, we use Hoeffdings inequality to determine the number of input data samples for given α and ϵ. In our experiments, we fix ϵ to be 0.03 and vary α to be 0.95, 0.99, and 0.999, which requires at least 2049, 2943 and 4222 samples, respectively. In our evaluation, we draw random test inputs from a Gaussian distribution with μ=5.0 and σ=3.0. Each bitwidth reduction is verified by HeteroRefactor for an acceptable loss (e) ($10^{-2}, 10^{-4}, or 10^{-6}$), where the acceptable loss indicates the number of accuracy digits.

Table 4 summarizes the probabilistic verification results for different α and e configurations. For each configuration, we report verification results for 8 and 16 bits floating point where `N` indicates verification failure, and the column HR reports the smallest verified bitwidth. As expected, a higher precision requirement and higher confidence requires higher FP bitwidth. The results show that 32 bit floating point could be reduced to 24bit with an acceptable loss of 0.01 at 99.9% confidence level.

We then synthesize the refactored program using Vivado HLS 2018.03 targeting a Xilinx XCVU9P FPGA. Rows F9-F10 in Table 1 summarize the resource utilization for each subject program. The `Orig` row indicates the original program with 32-bit float type and α represents the confidence probability. Then we report the resource utilization of FF, LUT, DSP for each combination of α and accuracy loss e. HeteroRefactor can achieve up to 38.2% reduction in FF, 28.6% reduction in LUT, and 50% decrease in DSP. Such reduction affects DSP usage significantly because floating elements are mapped to DSPs. As existing HLS flow does not support arbitrary floating-point type, so we could not find any hand-optimized kernels, and thus we can only compare against the default high bitwidth version.

Summary 3

HeteroRefactor lowers floating-point bitwidths from 32 bits to custom 24 bits, while providing probabilistic guarantee that the precision loss is within the acceptable loss of 0.01 at 99.9% confidence level. This reduces DSP usage on FPGA almost by half.

5 RELATED WORK

Automated Refactoring. Since pioneering work on automated refactoring in the early 90s [27, 42, 45], recent studies find that real-world refactorings are generally not semantics-preserving [36, 37], are done manually [63], are error-prone [35, 44], and are beyond the scope and capability of existing refactoring engines. A recent study with professional developers finds that almost 12% of refactorings are initiated by developers' motivation to improve performance. HeteroRefactor builds on this foundation of automated refactoring [42] but repurposes it to improve performance in the new era of heterogeneous computing with re-programmable circuits and to lower the FPGA programming barrier for software engineers. While HeteroRefactor's bitwidth reduction code transformation and removal of recursion and pointers is not semantics-preserving, HeteroRefactor guarantees the overall semantics-preservation by leveraging selective offloading from CPU to FPGA in tandem.

Dynamic Invariant. Determining program invariants has been explored widely using both static and dynamic techniques. HeteroRefactor is inspired by Daikon [21], which generates invariants of 22 kinds for C/C++/Java programs. Kataoka et al. [34] detect the symptoms of a narrow interface by observing dynamic invariants and refactors the corresponding API. Different from these, HeteroRefactor aims to enable regular software developers to program FPGA, leverages selective offloading in tandem to guarantee correctness, and does not require having representative data apriori.

In addition to running existing tests, a developer may use systematic test generation tools [23, 24, 57] or test minimization [32, 61] to infer FPGA-specific invariants, as representative data is not required for correctness. For example, one may use a white-box testing technique that models individual program paths through symbolic execution [28, 51] and sample data to achieve high path coverage [32, 61] or use the stratified sampling methods [1, 25].

HLS Optimization. Klimovic et al. [38] optimize FPGA accelerators for common-case inputs by reducing bitwidths using both bitmask analysis and program profiling [26]. When inputs exceed the common-case range, a software fallback function is automatically triggered. Their simulation results estimate that an accelerator's area may be reduced by 28% on average. While their approach is similar to HeteroRefactor, its scope is limited to monitoring integer values and they do not implement a systematic approach to monitor bitwidth invariants and the size of a recursive data structure at the kernel level, nor automatically assess the impact of tuning variable-width floating-point precision with a given error bound. Unlike our results in Section 4 which presents real hardware execution results on Xilinx Virtex UltraScale+ XCVU9P FPGA, they only present estimated software simulation results. To our knowledge, HeteroRefactor is the *first tool* for heterogeneous computing with FPGA that incorporates dynamic invariant analysis, automated kernel refactoring, selective offloading, and synthesized FPGA.

Several approaches provide HLS libraries for implementing variable width floating-point computation units and recursive data structures, but leave it to the programmer to specify which parameters to use and to rewrite their kernel code manually. For example, Thomas [62] presents an HLS backend for generating a customized floating-point accelerator using C++ template-based, parameterized types. However, this approach requires the user to manually specify the bitwidths for an exponent and fraction, which is automated in HeteroRefactor. As another example, SynADT [68] is an HLS library for representing recursive data structures such as linked lists, binary trees, hash tables, and vectors from pointers, and its implementation internally uses arrays and a shared system-wide memory allocator instead. However, SynADT supports only a limited set of data structures and does not address the developer burden of manually refactoring their kernel code. Different from SynADT, HeteroRefactor automatically monitors the appropriate size for a recursive data structure and performs fully automated kernel transformation to convert pointer usage to operations on a finite-sized array and implements a guard-condition based offloading. Our empirical evaluation demonstrates that, indeed, HeteroRefactor can significantly reduce resource utilization for most cases by coming up with a tighter bound for the required size of a recursive data structure, compared to the typical approach where a user has to over-approximate and over-provision resource usage in FPGA.

HeteroRefactor differs from static analysis methods which results in over-approximation. For example, Bitwise [60] propagates bitwidth constraints to variables based on the flow graph of bits. MiniBit [41] minimizes integer and fixed-point data signals with a static method based on affine arithmetic. Cong et al. [15] uses affine arithmetic, general interval arithmetic and symbolic arithmetic methods to optimize for fixed-point data.

HeteroRefactor uses an ahead-of-time profiling phase, in contrast with JIT compilation techniques [4], because FPGA synthesis (i.e., baking hardware) often takes a couple days, justifying and motivating thorough dynamic invariant analysis and FP probabilistic verification before FPGA synthesis.

Tuning Floating-point Precision. Precimonious [54] is a floating point precision tuning tool that uses dynamic analysis and delta-debugging to identify lower precision instruction that satisfies the user-specified acceptable precision loss constraint. HeteroRefactor's FP tuning is inspired by the success of Precimonious. However, HeteroRefactor extends this idea by adding a probabilistic verification logic to provide statistical guarantee on precision loss. While Pecimonious is a software-only analysis tool for FP tuning, HeteroRefactor is an end-to-end approach that integrates dynamic invariant analysis, automated refactoring, and FPGA synthesis.

6 CONCLUSION

Traditionally, automated refactoring has been used to improve software maintenability. To meet increasing demand for developing new hardware accelerators and to enable software engineers to leverage heterogeneous computing environments, we adapt and expand the scope of automated refactoring. HeteroRefactor provides a novel, end-to-end solution that combines (1) dynamic analysis for identifying common-case sizes, (2) kernel refactoring to enhance HLS synthesizability and to reduce on-chip resource usage on FPGA, and (3) selective offloading with guard checking to guarantee correctness. For the transformed recursive programs, HeteroRefactor reduces BRAM by 85% and increases frequency by 65%. For integer optimization, it reduces the number of bits for integers by 76%, leading to 41% decrease in BRAM. For floating-point optimization, it reduces DSP usage by 50%, while guaranteeing that the precision loss is within 0.01 with 99.9% confidence.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/2465351.2465355

[2] Alfred V Aho and Margaret J Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM* 18, 6 (1975), 333–340.

[3] Amazon.com. 2019. Amazon EC2 F1 Instances: Run Custom FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1. (2019).

[4] Matthew Arnold, Stephen J Fink, David Grove, Michael Hind, and Peter F Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 93, 2 (2005), 449–466.

[5] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*. Springer, 132–146.

[6] Uday Kumar Bondhugula. 2008. *Effective automatic parallelization and locality optimization using the polyhedral model*. Ph.D. Dissertation. The Ohio State University.

[7] Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.".

[8] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2015. Expanding OpenFlow capabilities with virtualized reconfigurable hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 94–97.

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

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[11] Andrew A Chien, Allan Snavely, and Mark Gahagan. 2011. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science* 4 (2011), 1987–1996.

[12] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?. In *2010 43rd annual IEEE/ACM international symposium on microarchitecture*. IEEE, 225–236.

[13] 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).

[14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the 51st Annual Design Automation Conference*. ACM, 1–6.

[15] Jason Cong, Karthik Gururaj, Bin Liu, Chunyue Liu, Zhiru Zhang, Sheng Zhou, and Yi Zou. 2009. Evaluation of static analysis techniques for fixed-point precision optimization. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 231–234.

[16] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software infrastructure for enabling FPGA-based accelerations in data centers. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 154–155.

[17] 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.

[18] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*. IEEE, 1–6.

[19] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Langlois. 2019. Module-per-Object: a Human-Driven Methodology for C++-based High-Level Synthesis Design. *arXiv preprint arXiv:1903.06693* (2019).

[20] Rene De La Briandais. 1959. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*. ACM, 295–298.

[21] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[22] 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.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[24] Patrice Godefroid, Michael Y. Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society. http://www.truststc.org/pubs/499.html

[25] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 383–397. https://doi.org/10.1145/2694344.2694351

[26] Marcel Gort and Jason H Anderson. 2013. Range and bitmask analysis for hardware optimization in high-level synthesis. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 773–779.

[27] William G. Griswold. 1991. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. Dissertation. University of Washington.

[28] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. 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)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/3338906.3338953

[29] Prabhat Gupta. 2019. Xeon+FPGA Platform for the Data Center. https://www.archive.ece.cmu.edu/~calcm/carl/lib/\exe/fetch.php?media=carl15-gupta.pdf. (2019).

[30] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 1192–1195.

[31] Wassily Hoeffding. 1994. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*. Springer, 409–426.

[32] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A General Framework and Tool for Supporting Test-suite Minimization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 419–429. https://doi.org/10.1109/ICSE.2009.5070541

[33] Steven Huss-Lederman, Elaine M Jacobson, Jeremy R Johnson, Anna Tsao, and Thomas Turnbull. 1996. Implementation of Strassen's algorithm for matrix multiplication. In *Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. IEEE, 32–32.

[34] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. 2001. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*. Florence, Italy, 736–743.

[35] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of Refactorings during Software Evolution. In *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*.

[36] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[37] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 1–1. https://doi.org/10.1109/TSE.2014.2318734

[38] Ana Klimovic and Jason H Anderson. 2013. Bitwidth-optimized hardware accelerators with software fallback. In *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 136–143.

[39] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Ieee, 115–127.

[40] Henry Oliver Lancaster and Eugene Seneta. 2005. Chi-square distribution. *Encyclopedia of biostatistics* 2 (2005).

[41] Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. 2005. MiniBit: bit-width optimization via affine arithmetic. In *Proceedings of the 42nd annual Design Automation Conference*. ACM, 837–840.

[42] Tom Mens and Tom Tourwe. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[43] Giovanni De Micheli. 1994. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education.

[44] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 287–297.

https://doi.org/10.1109/ICSE.2009.5070529

[45] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph.D. Dissertation. University of Illinois, Urbana-Champaign, IL, USA. citeseer.ist.psu.edu/opdyke92refactoring.html

[46] James L Peterson and Theodore A Norman. 1977. Buddy systems. *Commun. ACM* 20, 6 (1977), 421–431.

[47] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. 2015. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 157–162.

[48] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays.* ACM, 29–38.

[49] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[50] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[51] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15).* ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/2815400.2815418

[52] 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).* IEEE, 110–119.

[53] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. 2016. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 1074–1085.

[54] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–12.

[55] Giacinto Paolo Saggese, Antonino Mazzeo, Nicola Mazzocca, and Antonio GM Strollo. 2003. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *International Conference on Field Programmable Logic and Applications.* Springer, 292–302.

[56] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *PLDI.*

[57] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13).* ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[58] Nitish Srivastava, Steve Dai, Rajit Manohar, and Zhiru Zhang. 2017. Accelerating Face Detection on Programmable SoC Using C-Based Synthesis. In *25^{th} ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* https://doi.org/10.1145/3020078.3021753

[59] Richard Stallman, Roland Pesch, Stan Shebs, et al. 2002. Debugging with GDB. *Free Software Foundation* 51 (2002), 02110–1301.

[60] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bidwidth Analysis with Application to Silicon Compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00).* ACM, New York, NY, USA, 108–120. https://doi.org/10.1145/349299.349317

[61] Sriraman Tallam and Neelam Gupta. 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05).* ACM, New York, NY, USA, 35–42. https://doi.org/10.1145/1108792.1108802

[62] David B Thomas. 2019. Templatised Soft Floating-Point for High-Level Synthesis. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE.

[63] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on.* 233 –243. https://doi.org/10.1109/ICSE.2012.6227190

[64] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management.* Springer, 1–116.

[65] Xilinx. 2019. UltraScale Architecture and Product Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. (2019).

[66] Xilinx. 2019. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. (2019).

[67] Xilinx. 2019. Xilinx Virtex UltraScale+ FPGA VCU1525. https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html. (2019).

[68] 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).* IEEE, 64–71.

[69] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: an accelerator automation framework for heterogeneous computing in datacenters. In *Proceedings of the 55th Annual Design Automation Conference (DAC).* ACM, 153.

[70] 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. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb 2018).