HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA

Jason Lau*, Aishwarya Sivaraman*, Qian Zhang*, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim
University of California, Los Angeles
{lau, dcssiva, zhangqian, gulzar, cong, miryung}@cs.ucla.edu
*Equal co-first authors in alphabetical order

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, it selectively offloads the computation from CPU to FPGA, only if an input falls within the dynamic invariant. On average, for a recursive program of size 175 LOC, an expert FPGA programmer would need to write 185 more LOC to implement an HLS compatible version, while HeteroRefactor automates such transformation. Our results on Xilinx FPGA show that HeteroRefactor minimizes BRAM by 83% and increases frequency by 42% for recursive programs; reduces BRAM by 41% through integer bitwidth reduction; and reduces DSP by 50% through floating-point precision tuning.

KEYWORDS
heterogeneous computing, automated refactoring, FPGA, high-level synthesis, dynamic analysis
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. 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 resource usages and increases the frequency at the FPGA level.

We evaluate HeteroRefactor on ten programs, including five handwritten recursive programs, three integer-intensive programs from Rosetta benchmark [84], and two floating-point-intensive programs from OpenCV [6]. We generate kernels targeting to a Xilinx Virtex UltraScale+ XCVU9P FPGA on a VCU1525 Reconfigurable Acceleration Platform [80] 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 175 LOC, an expert FPGA programmer would need to write 185 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 83% decrease in BRAM and 42% 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 (FF), 21% reduction in look-up tables, 41% reduction in BRAM, and 52% decrease in DSP.

3. For floating-points, HeteroRefactor automatically reduces the bitwidth while providing a probabilistic guarantee for a user-specific quality loss and confidence level. The optimized accelerator can achieve up to 61% reduction in FF, 39% reduction in LUT, and 50% decrease in DSP when an acceptable precision loss is specified as $10^{-4}$ at 95% confidence level.

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 to improve 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.

HeteroRefactor’s source code and experimental artifacts are publicly available at https://github.com/heterorefactor/heterorefactor.

## 2 BACKGROUND

This section overview 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), and millions of flip-flop registers (FFs) [78]. Each k-input LUT can implement any Boolean function up to k inputs. An FPGA must 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 [21, 28, 50]. 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. For example, even for tens of lines of code, hardware synthesis can take hours for our subjects in Section 4.

Therefore, such long hardware synthesis time justifies the cost of manual rewriting of kernels for optimized resource allocation, frequency, and power utilization. In other words, this motivates HeteroRefactor to invest time in a-priori dynamic analysis as opposed to just-in-time compilation to optimize FPGA, 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 [25]. A developer must have interdisciplinary expert knowledge in both hardware and software and know obscure platform-dependent details [15]. 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 [21, 79], which is the most widely used FPGA HLS in the community, although such techniques can be easily generalized to other HLS tools, such as Intel HLS Compiler, Catapult HLS from Mentor, and CyberWorkBench from NEC.

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 reinterpretation 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 recursions. 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. 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—e.g., #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, 23, 56, 85].

Optimization of data movement. Accessing of the device memory can be more efficient by packing bits into the width of DRAM access of 512 bits. To overlap communication with computation, a developer could explicitly implement a double buffering technique [15]. To cache data, developers need to explicitly store them on chip through data tiling, batching or reusing [10, 56, 65].

Reducing resource consumption. Provisioning more processing elements or a larger cache will require using more on-chip 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 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 illustrates our new contributions, highlighted with **bold** and **red**, relative to the prior HLS literature. There exists many automated approaches for generating device and host interfaces [20, 61, 83], exploring parallelization opportunities [24, 34, 46, 83],

---

**Related Work**

**Traditional Workflow**

1. **C/C++ Program**
   - Rewrite by expert
   - Synthesizable FPGA Code
   - Rewrite Code

**HETERORefactor Workflow**

1. **C/C++ Program**
   - Input Data
   - Dynamic Analysis and Software Refactoring
   - Resource Efficient FPGA Code
   - Selective Offloading to FPGA

---

**Figure 1:** Overview of refactoring for high-level synthesis.

**Figure 2:** Approach overview of HETERORefactor.
and optimizing data movement [10, 20, 24, 46, 55, 56]. But general methods for reducing resource consumption, pointer support, memory management and recursion support remain as open research questions and no automated kernel factoring 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 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). All three refactorings are based on the insight that a-priori dynamic analysis improves FPGA synthesizability and resource efficiency and that dynamic, input-dependent offloading can guarantee correctness. Figure 3 details the three components that work in concert: (A) instrument for FPGA-specific dynamic invariant analysis, (B) source-to-source transformation using 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 is within a given 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 4a: *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 a source to source compiler framework, ROSE [59] to instrument code for identifying the size of recursive data structures and the corresponding stack depth and performs source-to-source transformation based on the size.

- **3.1.1 Refactoring-based Instrumentation.** **HeteroRefactor** instruments memory allocation and de-allocation function calls (e.g., allocation of a linked list node), and adds tracing points at the entry and exit of recursive functions to monitor a stack depth. **HeteroRefactor** then determines the number of elements allocated for each data structure based on the collected sizes. In Figure 4a, **HeteroRefactor** sets a tracing point at line 3 to record the number of allocated nodes and another tracing point at line 16 to record the released count. To monitor the recursion depth, **HeteroRefactor** inserts tracing points call at the function entry point and re at the function exit point of each recursive function. In Figure 4a, call is inserted before line 6, and re is inserted at line 6 and after line 9. **HeteroRefactor** then maintains a variable stack size for each function, which is incremented every time the program reaches call and decremented when it reaches re. 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 ROSE [59] 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 on-chip memory allocation, removes all pointers except for those with native
Figure 4: Example of recursive data structures: binary tree.

(a) Original kernel code using pointers and memory allocation.

```c
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   traverse(curr->left);
8   traverse(curr->right);
9   void top(int n, int *output_if) { 
10  int values[3] = {5, 4, 3};
11  insert(&root, 3, values);
12  Node *root; init(&root); // ...
```

(b) Refactored kernel code (schematic).

```c
1 bool guard_error = false;
2 struct Node { Node_ptr left, right; int val; };
3 struct Node_ptr; // buddy allocation
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)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   stackcontext = sTRAVERSAL_STACK_SIZE, (curr->curr.loc[0]);
13   while (ts.empty()) (curr = *s.pop(); goto t.c.loc);
14   if (curr->curr->curr) continue;
15   if (curr->curr->curr->curr) continue;
16   if (curr->curr->curr->curr->curr) continue;
17   c = curr[1].data[1]; s.push(curr->curr[1].data[1]); c = curr[1].data[1];
18   h = (guard_error=true; return;)
19  l1: // traverse(right) ... 
20  l2: // traverse(left) ...
21  void top(int n, int *output_if, bool *fail) { 
22  int values[3] = {5, 4, 3};
23  insert(&root, 3, values);
24  Node_ptr root; init(&root); // ...
25  Node_ptr root; init(&root); // ...
27  Node_free(root); *fail = guard_error; }
```

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 4b). The per-type allocation strategy with an array is based on two reasons—HLS only supports pointer reinterpretation 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 [54] and allocate from the array. The buddy memory system requires less overhead and has little external fragmentation [77], 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 4b) 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 4b). 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 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 4b), typecasting (line 8 Figure 4b), and function parameters (line 10 Figure 4b) 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 indirect expression (ptr->) and structure dereference operators (ptr->.field) into array accesses with pointer integer as the array index. Similarly, the subscript operators (ptr[i]) 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 4a) to array access (line 15 in Figure 4b).

Rule 3: Convert Recursion to Iteration. To transform recursive functions into non-recursive ones, we create a stack (line 12 Figure 4b) for each function with all local variables. 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 4b). 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 4b), then continue to the first line of the function (line 18 Figure 4b). 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 Figure 4b).

3.2 Integer

3.2.1 Kvasir-based Instrumentation. Daikon is a dynamic invariant detection tool [27] 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 state information by running the program. FPGA kernels are programmed in C/C++ for HLS; hence, we use Kvasir [27], 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 a variable bitwidth leads to resource reduction in FPGA directly [45].
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 5a. 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 Refactoring. Rule Modify Variable Type. To convert an integer to an arbitrary precision integer, we leverage ap_uint<k> or ap_int<k> provided by Vivado HLS, which defines an arbitrary precision integer of k bits. As an example, the input haar_counter to method weakClassifier in Figure 5a 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. It parses the program’s AST using ROSE [59], identifies the variable declaration node for stddev, coord, haarC, and w_id, and then modifies the corresponding type as shown in Figure 5b.

3.3 Floating Point

Unlike the reduction of integer bitwidth in Section 3.2, reducing the 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 bitwidths, (2) estimation of the required number of input data samples based on Hoeffding’s inequality [37], (3) test generation and differential execution, and (4) probabilistic verification for FP errors.

Prior work on reducing FP precision in CPU [62, 63] 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 $\epsilon$ with probability $p$. 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 variables, constants and operations. For each input $i \in I$, we compute the actual error between the high and the low bitwidth variants, $hp(i) - lp(i)$. We then check whether this FP error is within the acceptable precision loss $\epsilon$, indicated by a predicate $c_i = (hp(i) - lp(i) < \epsilon)$ which forms a distribution $B$. When the empirical measurement $\overline{c}_I$ of the given input samples is higher than the target probability $p$, the verification is passed.

HeteroRefactor takes as inputs: (1) a program, (2) a set of sampled inputs $I$ or a statistical distribution, (3) an acceptable loss (error) $\epsilon$, (4) a required probability $p$, (5) a required confidence level $1 - \alpha$ and (6) deviation $e$. We use Hoeffding’s inequality [37] to compute the minimum number of samples required to satisfy the given confidence level $1 - \alpha$ and deviation $e$. Equation 1 shows the probability that the empirical measurement $\overline{c}_I$ of the distribution $B$ deviates from its actual expectation $E[\overline{c}]$ by $\epsilon$, which should be less than $\alpha$ to achieve our target. Similar to Sampson et al.’s probabilistic assertion [66], 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.

\[
P[|\overline{c}_I - E[\overline{c}]| \geq \epsilon] \leq 2e^{-2n\epsilon^2}
\]

\[
n \geq \ln(2/\alpha)/(2\epsilon^2)
\]
For example, when a user wants the actual FP error between the original version (Figure 6a) and the low precision variant (Figure 6b) to be less than $10^{-4}$ with 95% probability, 95% confidence level and 0.03 deviation, the minimum number of samples required is 2049.

During differential testing with respect to input $i$, if the proportion $\overline{c}$ of passing samples to $|I|$ is greater than $p$, 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 $\text{l2norm}$ in Figure 6a, HeteroRefactor traverses its AST and redefines the type of variable query, data, and dim originally declared as $\text{float}$ using $\text{thls::fp}$ _$\text{flopoco}$E, F_, whose library is based on Thomas’ work on *templated soft floating-point type for HLS* [75]. $E$ is the number of exponent bits and $F$ is the number of fractional bits (excluding 1 implicit bit). For example, $\text{thls::fp}_{-}\text{flopoco}3<8, 23>$ is 32 bit $\text{float}$ type, and $\text{thls::fp}_{-}\text{flopoco}5, 16>$ 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 $\text{thls::fp}$ _$\text{flopoco}$E, F_, subtraction is not supported [75]. Hence, we convert subtraction in $\text{l2norm}$ (line 4 in Figure 6a) to corresponding neg and add, i.e., $\text{subtract}(a, b) = \text{add}(a, \text{neg}(b))$ using a variable $\text{fp}_{-}\text{neg}_{-}1$ to store the intermediate result (lines 7-8 in Figure 6b).

**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 6b). This involves adding code to invoke the original and generated low precision variants 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 4b, we insert a guard condition at $\text{Node\_malloc}$. The condition sets a global variable guard$_{\text{error}}$ 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 5b, 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 5b). For this, we first identify all expressions containing the reduced bitwidth variables, and if the expression contains binary operations, we insert a guard.

### 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 [26] 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 [40] is a recursive algorithm for matrix multiplication. For integer optimization, we use face detection and 3D rendering from Rosetta [69, 84] (b6 and b7). We also write (b8) bubble sort. For FP bitwidth reduction, we modify two programs—(F9) KNN-L2norm and (F10) RGB2YUV from OpenCV examples [6].

These subject programs demonstrate HeteroRefactor’s capability on improving synthesizability and resource efficiency. For recursive data structures, the original programs are not synthesizable and cannot run on FPGA prior to our work. Thus, we compare our results against manually ported code in terms of human effort and resource utilization. The hand-optimized programs are written by experienced graduate students from an FPGA research group at UCLA. Original programs for integer and floating-point can already run on FPGA. For integers, we compare resource utilization to both original (unoptimized) and manually optimized programs, which are directly from Rosetta [69, 84]. For floating-point, there is no comparison with hand-optimized versions, because a manual optimization attempt will be similar to the verification procedure of HeteroRefactor.

Though the code size of subject programs looks small, these programs are rather sizable compared against well-known FPGA HLS benchmarks [36, 60]. 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. The automated refactoring is implemented based on ROSE compiler’s version 0.9.11.0. The refactored programs are synthesized to RTL to estimate the resource utilization by Vivado Design Suite 2018.03. The 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 recursive data structure programs are now synthesizable using HeteroRefactor that fail compilation with Vivado HLS. For RQ2, we measure manual porting effort as LOC and characters in the code. For RQ3, we assess reduction in resource utilization and increase in frequency of the resulting FPGA
Table 1: Resource utilization for HeteroRefactor

<table>
<thead>
<tr>
<th>ID/Program</th>
<th>#LUT</th>
<th>#FF</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1/ Aho-Corasick</td>
<td>Orig</td>
<td>Not Synthesizable</td>
<td>3287</td>
<td>4666</td>
</tr>
<tr>
<td>HR-8K</td>
<td>5492</td>
<td>5085</td>
<td>678</td>
<td>10</td>
</tr>
<tr>
<td>HR-2K</td>
<td>5234</td>
<td>5006</td>
<td>206</td>
<td>10</td>
</tr>
<tr>
<td>R2/ DFS</td>
<td>Orig</td>
<td>Not Synthesizable</td>
<td>1471</td>
<td>1961</td>
</tr>
<tr>
<td>HR-8K</td>
<td>2634</td>
<td>2901</td>
<td>254</td>
<td>0</td>
</tr>
<tr>
<td>HR-2K</td>
<td>2563</td>
<td>2881</td>
<td>69</td>
<td>0</td>
</tr>
<tr>
<td>R3/ Linked List</td>
<td>Orig</td>
<td>Not Synthesizable</td>
<td>2993</td>
<td>3732</td>
</tr>
<tr>
<td>HR-8K</td>
<td>3771</td>
<td>4044</td>
<td>318</td>
<td>0</td>
</tr>
<tr>
<td>HR-2K</td>
<td>3655</td>
<td>3936</td>
<td>83</td>
<td>0</td>
</tr>
<tr>
<td>R4/ Merge Sort</td>
<td>Orig</td>
<td>Not Synthesizable</td>
<td>2755</td>
<td>2878</td>
</tr>
<tr>
<td>HR-8K</td>
<td>2751</td>
<td>2958</td>
<td>367</td>
<td>0</td>
</tr>
<tr>
<td>HR-2K</td>
<td>2603</td>
<td>2951</td>
<td>105</td>
<td>0</td>
</tr>
<tr>
<td>R5/ Strassen’s</td>
<td>Orig</td>
<td>Not Synthesizable</td>
<td>21631</td>
<td>13722</td>
</tr>
<tr>
<td>HR-8K</td>
<td>20303</td>
<td>14899</td>
<td>223</td>
<td>12</td>
</tr>
<tr>
<td>HR-2K</td>
<td>19591</td>
<td>14634</td>
<td>68</td>
<td>12</td>
</tr>
<tr>
<td>I6/ Face Detection</td>
<td>Orig</td>
<td>Manual</td>
<td>11325</td>
<td>5784</td>
</tr>
<tr>
<td>HR</td>
<td>10298</td>
<td>4770</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>I7/ 3D Rendering</td>
<td>Orig</td>
<td>Manual</td>
<td>3828</td>
<td>2033</td>
</tr>
<tr>
<td>HR</td>
<td>2239</td>
<td>1357</td>
<td>67</td>
<td>12</td>
</tr>
<tr>
<td>I8/ Bubble Sort</td>
<td>Orig</td>
<td>Manual</td>
<td>316</td>
<td>125</td>
</tr>
<tr>
<td>HR</td>
<td>306</td>
<td>125</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F9/ KNNl2norm</td>
<td>Orig</td>
<td>Manual</td>
<td>88843</td>
<td>18591</td>
</tr>
<tr>
<td>p</td>
<td>e = 10^{-2}</td>
<td>0.95</td>
<td>80163</td>
<td>15257</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.99</td>
<td>82228</td>
<td>15626</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.999</td>
<td>82228</td>
<td>15626</td>
</tr>
<tr>
<td>p</td>
<td>e = 10^{-4}</td>
<td>0.95</td>
<td>88952</td>
<td>17102</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.99</td>
<td>88952</td>
<td>17102</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.999</td>
<td>88952</td>
<td>17855</td>
</tr>
<tr>
<td>p</td>
<td>e = 10^{-6}</td>
<td>0.95</td>
<td>88843</td>
<td>18591</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.99</td>
<td>88843</td>
<td>18591</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.999</td>
<td>88843</td>
<td>18591</td>
</tr>
<tr>
<td>F10/ RGB2YUV</td>
<td>Orig</td>
<td>Manual</td>
<td>398444</td>
<td>73437</td>
</tr>
<tr>
<td>p</td>
<td>e = 10^{-4}</td>
<td>0.95</td>
<td>243516</td>
<td>28379</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.99</td>
<td>250044</td>
<td>28827</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.999</td>
<td>250044</td>
<td>28827</td>
</tr>
<tr>
<td>p</td>
<td>e = 10^{-5}</td>
<td>0.95</td>
<td>304956</td>
<td>49468</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.99</td>
<td>304956</td>
<td>49468</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.999</td>
<td>311532</td>
<td>49964</td>
</tr>
<tr>
<td>p</td>
<td>e = 10^{-6}</td>
<td>0.95</td>
<td>372236</td>
<td>66381</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.99</td>
<td>398444</td>
<td>73437</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.999</td>
<td>398444</td>
<td>73437</td>
</tr>
</tbody>
</table>

Table 2: Recursive data structure kernels, no extra code with HeteroRefactor v.s. effort for manual refactoring

<table>
<thead>
<tr>
<th>ID/Program</th>
<th>LOC</th>
<th>Manual</th>
<th>#LUT</th>
<th>Manual</th>
<th>Δ LOC</th>
<th>Manual</th>
<th>Δ Chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1/A.-C.</td>
<td>190</td>
<td>291</td>
<td>5673</td>
<td>8776</td>
<td>35%</td>
<td>105</td>
<td>8776</td>
</tr>
<tr>
<td>R2/DFS</td>
<td>86</td>
<td>198</td>
<td>2236</td>
<td>5699</td>
<td>61%</td>
<td>105</td>
<td>5699</td>
</tr>
<tr>
<td>R3/L. List</td>
<td>131</td>
<td>235</td>
<td>3061</td>
<td>6686</td>
<td>54%</td>
<td>105</td>
<td>6686</td>
</tr>
<tr>
<td>R4/M. Sort</td>
<td>128</td>
<td>342</td>
<td>3267</td>
<td>9124</td>
<td>64%</td>
<td>105</td>
<td>9124</td>
</tr>
<tr>
<td>R5/Strassen’s</td>
<td>342</td>
<td>735</td>
<td>10026</td>
<td>40971</td>
<td>76%</td>
<td>105</td>
<td>40971</td>
</tr>
<tr>
<td>Geomean</td>
<td>49%</td>
<td></td>
<td>56%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

which harms timing.

To evaluate reduction in resource utilization, we instrument the programs using randomly generated input data with typical size of 1k, 2k, 4k or 8k. 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 and optimize the programs in Table 2 for a conservative data structure 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 the hand-optimized 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, there is 83% reduction on average in BRAM, compared to the manually refactored program with a conservative array size. This decrease is significant because Vivado HLS stores most of the large array in BRAM. On the other hand, there is an increase of 302 units in LUT and 494 units in FF on average compared to the hand-optimized version. This small overhead is caused by the fixed-sized buddy memory system which does not increase, as the user design scales.

We implement FPGA accelerator on-board with a target frequency of 300 MHz. Figure 7 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: $F_{max} = 1/(1/(300MHz + WNS))$. If the input recursive data structure size is 2k, on average, there is 42% increase in frequency compared to the hand-written code with a conservative size of 16k. 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.
We assess the hypothesis that reducing bitwidth based on dynamic invariants leads to reduction in resource utilization for integers. We measure 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 use dynamic invariants (Table 3) to create a bitwidth optimized program (e.g., Figure 5b). 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 [69] and resize them to the dimension of 16x16. The program uses pre-trained weights declared invariants (Table 3) to create a bitwidth optimized program (e.g., Figure 5b).

We evaluate the effectiveness of HeteroRefactor in providing a probabilistic guarantee while lowering a bitwidth, and reducing 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 a user given loss, ε, with probability p. As described in Section 3.3, we use Hoeffdings inequality to determine the number of input data samples for given (1 − α) and ε. In our experiments, we fix ε to be 0.03, vary p to be 0.95, 0.99, and 0.999, and keep the confidence level the same as p, i.e., (1 − α) = p, which require at least 2049, 2943 and 4222 samples, respectively. We also consider the input features of FPGA kernels to determine the final number of samples. For example, RGB2YUV requires that the inputs must be multiples of 16, so the final number of samples is 2064 rather than 2049 when p is 0.95. In our evaluation, we draw random test inputs within 0 to 255. The bitwidth reduction is verified by HeteroRefactor with an acceptable loss (ε) (10^{-2}, 10^{-4}, or 10^{-6}) for KNN-l2norm and an acceptable loss (ε) (10^{-4}, 10^{-5}, or 10^{-6}) for RGB2YUV.

Table 4 summarizes the probabilistic verification results for different ε and p configurations. For each configuration, we report verification results for 8 and 16 bits floating-point, where N indicates a verification failure, and the column HR reports the smallest verified bitwidth. As expected, a higher precision and confidence requirement leads to a higher FP bitwidth. The results show that a
represents the probability and the confidence level. We report the resource utilization of FF, LUT, and DSP for each program. Table 5 summarizes the instrumentation overhead and refactoring overhead compared against the synthesis time of the original programs. For recursive data structures, both the instrumentation and refactoring overhead are less than 1%. For integers, our refactored kernels are slower than CPU, because there is no instrumentation required. HeteroRefactor induces less than 2% overhead on floating-point bitwidth tuning.

We compare the execution performance of the refactored kernel against running the original program on CPU. For floating-point programs, our experiment shows a significant speedup up to \( \times 61 \) reduction in FF, 39% reduction in LUT, and 50% decrease in DSP. 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.

Table 4: Probabilistic floating-point verification

<table>
<thead>
<tr>
<th>Program</th>
<th>( p )</th>
<th>( e = 10^{-4} )</th>
<th>10^{-4}</th>
<th>10^{-6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>F9/KNN</td>
<td>0.95</td>
<td>83.5 / 1.8%</td>
<td>54.5 / 1.2%</td>
<td>12.8 / 0.3%</td>
</tr>
<tr>
<td>0.99</td>
<td>58.2 / 0.3%</td>
<td>30.4 / 0.2%</td>
<td>11.7 / 0.1%</td>
<td></td>
</tr>
<tr>
<td>0.999</td>
<td>60.8 / 0.3%</td>
<td>25.9 / 0.1%</td>
<td>12.8 / 0.1%</td>
<td></td>
</tr>
</tbody>
</table>

Summary 3

HeteroRefactor reduces the floating-point bitwidth while providing a probabilistic guarantee for a user-specified quality loss, probability and confidence level. It can achieve up to 61% reduction in FF, 39% in LUT, and 50% in DSP.

4.4 Overhead and Performance

Table 5 summarizes the instrumentation overhead and refactoring overhead for R1-I8 compared against the synthesis time of the original programs. For recursive data structures, both the instrumentation and refactoring overhead are less than 1%. For integers, HeteroRefactor induces less than 1% refactoring overhead, and its instrumentation overhead comes from Kvasar. For floating-point programs F9-F10, Table 6 summarizes the differential execution overhead compared against the synthesis time of the original programs with a specific quality loss \( e \) and probability \( p \), because there is no instrumentation required. HeteroRefactor induces less than 2% overhead on floating-point bitwidth tuning.

Table 5: Runtime overhead for recursions and integers

<table>
<thead>
<tr>
<th>Program</th>
<th>Instrumentation</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (min)</td>
<td>ratio</td>
</tr>
<tr>
<td>R1/Aho-Corasick</td>
<td>0.10</td>
<td>0.26%</td>
</tr>
<tr>
<td>R2/DFS</td>
<td>0.06</td>
<td>0.26%</td>
</tr>
<tr>
<td>R3/Linked List</td>
<td>0.12</td>
<td>0.49%</td>
</tr>
<tr>
<td>R4/Merge Sort</td>
<td>0.05</td>
<td>0.20%</td>
</tr>
<tr>
<td>R5/Strassen’s</td>
<td>0.09</td>
<td>0.20%</td>
</tr>
<tr>
<td>I6/Face Detection</td>
<td>0.15</td>
<td>0.62%</td>
</tr>
<tr>
<td>I7/3D Rendering</td>
<td>13.66</td>
<td>64.76%</td>
</tr>
<tr>
<td>I8/Bubble Sort</td>
<td>10^{-3}</td>
<td>0</td>
</tr>
</tbody>
</table>

32 bit floating-point variable could be reduced to using 21 bits with an acceptable loss of \( 10^{-4} \) at 95% confidence level.

We then synthesize the refactored program using Vivado HLS targeting a Xilinx XCVU9P FPGA. Rows F9-F10 in Table 1 summarize the resource utilization for each subject program. The 0.1g row indicates the original program with 32-bit float type and \( p \) represents the probability and the confidence level (1 - \( \alpha \)). Then we report the resource utilization of FF, LUT, and DSP for each combination of \( p \) and accuracy loss \( e \). HeteroRefactor can achieve up to 61% reduction in FF, 39% reduction in LUT, and 50% decrease in DSP. 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.

Table 6: Differential execution overhead for FP (sec / %)

<table>
<thead>
<tr>
<th>Program</th>
<th>( p )</th>
<th>( e = 10^{-4} )</th>
<th>10^{-4}</th>
<th>10^{-6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>F9/KNN</td>
<td>0.95</td>
<td>60.8 / 0.3%</td>
<td>29.4 / 0.2%</td>
<td>11.7 / 0.1%</td>
</tr>
<tr>
<td>0.99</td>
<td>58.2 / 0.3%</td>
<td>30.4 / 0.2%</td>
<td>11.7 / 0.1%</td>
<td></td>
</tr>
<tr>
<td>0.999</td>
<td>60.8 / 0.3%</td>
<td>25.9 / 0.1%</td>
<td>12.8 / 0.1%</td>
<td></td>
</tr>
<tr>
<td>R10/R2Y</td>
<td>0.95</td>
<td>83.5 / 1.8%</td>
<td>59.3 / 1.3%</td>
<td>13.7 / 0.3%</td>
</tr>
<tr>
<td>0.99</td>
<td>81.5 / 1.7%</td>
<td>58.5 / 1.2%</td>
<td>12.9 / 0.3%</td>
<td></td>
</tr>
<tr>
<td>0.999</td>
<td>82.3 / 1.7%</td>
<td>54.5 / 1.2%</td>
<td>13.7 / 0.3%</td>
<td></td>
</tr>
</tbody>
</table>

slower than running on CPU because I6 and I7 in Rosetta are designed to achieve higher energy efficiency but not higher processing throughput compared to CPU [84]. HeteroRefactor aims to reduce resource usage, while prior work [19, 24] achieves higher performance than CPU by leveraging more on-chip resources to achieve parallelism. HeteroRefactor could be used jointly with other tools to produce fast and resource-efficient FPGA accelerators.

5 RELATED WORK

Automated Refactoring. Since pioneering work on automated refactoring in the early 90s [32, 49, 53], recent studies find that real-world refactorings are generally not semantics-preserving [43, 44], are done manually [76], are error-prone [42, 51], 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 [44]. HeteroRefactor builds on this foundation [49] but repurposes it to improve performance in the new era of heterogeneous computing with re-programmable circuits. While HeteroRefactor’s refactoring is not semantics-preserving, it guarantees 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 [27], which generates invariants of 22 kinds for C/C++/Java programs. Kataoka et al. [41] detect the symptoms of a narrow interface by observing dynamic invariants and refactor the corresponding API. Different from these, HeteroRefactor does not require having representative data a-priori, as it leverages selective offloading to guarantee correctness. Therefore, a developer may use systematic test generation tools [29, 30, 67] or test minimization [38, 73] to infer FPGA-specific invariants, as representative data is not required for correctness.
**HLS Optimization.** Klomovic et al. [45] optimize FPGA accelerators for common-case inputs by reducing bitwidths using both bitmask analysis and program profiling [31]. 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. While we present real hardware 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, but leave it to the programmer to specify which parameters to use and to rewrite their kernel code manually. For example, Thomas [75] presents an HLS backend for generating a customized floating-point accelerator using C++ template-based, parameterized types. This approach requires the user to manually specify the bitwidths for an exponent and fraction, which is automated in HeteroRefactor.

HeteroRefactor differs from static analysis methods which results in over-approximation. For example, Bitwise [71] propagates bitwidth constraints to variables based on the flow graph of bits. MiniBit [48] minimizes integer and fixed-point data signals with a static method based on affine arithmetic. Cong et al. [18] uses affine arithmetic, general interval arithmetic and symbolic arithmetic methods to optimize for fixed-point data. In contrast to JIT compilation techniques [4], HeteroRefactor uses an ahead-of-time profiling phase, due to a long FPGA synthesis time.

**Recursion in Heterogeneous Computing.** Enabling recursive data structures in FPGA has been a long challenge because the address space for each array is separate in HLS/FPGA unlike traditional CPU architectures. SynADT [82] is an HLS library for representing linked lists, binary trees, hash tables, and vectors from pointers, and it internally uses arrays and a shared system-wide memory allocator [81]. However, SynADT supports only a limited set of data structures and requires developers to manually refactor. In contrast, HeteroRefactor automatically monitors an appropriate size of 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. Thomas et al. [74] use C++ templates to create a domain-specific language to support recursion in HLS. However, it requires extensive rewriting of control statements using lambdas.

Similar limitation existed on GPU with CUDA [52] and OpenCL [72]. For example, dynamic memory management on device global memory using malloc was not supported until CUDA 3.2 [52], and there is no implementation of malloc on shared-on-chip memory. However, one may write their own universal allocator for arbitrary types as a replacement for malloc on any memory [1, 39, 70] because GPU allows a single address space with regular access widths, similar to CPU, while FPGA does not. Such approaches [1, 39, 52, 70] still require manually specifying a heap size. HeteroRefactor automatically detects the required size of FPGA on-chip memory for recursive data structures using dynamic invariant detection, and fallbacks to CPU computation when the size invariants are violated.

**Tuning Floating-point Precision.** FPTuner [11] uses static analysis for automatic precision-tuning [68] of real valued expressions. It supports a single, double, or quadruple precision rather than an arbitrary-width FP type. Precimonious [63] 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 Precimonious 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 maintainability. To meet the 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 83% and increases frequency by 42%. 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 a user-specified precision loss of 0.01 with 99.9% confidence.

7 ACKNOWLEDGEMENT

We would like to thank Guy Van den Broeck, Brett Chalabian, Todd Millstein, Peng Wei, Cody Hao Yu and anonymous reviewers for their valuable feedback. We thank Janice Wheeler for proofreading the draft. This work is in part supported by NSF grants CCF-1764077, CCF-1527923, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, and Samsung grant. This work is also partially supported by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program and the contributions from the member companies under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program, including Xilinx and VMware. Jason Lau is supported by UCLA Computer Science Departmental Fellowship and Muhammad Gulzar is supported by Google PhD Fellowship.