2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

Design Patterns for Code Reuse in HLS Packet Processing Pipelines

Haggai Eran*†, Lior Zeno*, Zsolt István‡, and Mark Silberstein*
*Technion - Israel Institute of Technology †Mellanox Technologies ‡IMDEA Software Institute

***Abstract*—High-level synthesis (HLS) allows developers to be more productive in designing FPGA circuits thanks to familiar programming languages and high-level abstractions. In order to create high-performance circuits, HLS tools, such as Xilinx Vivado HLS, require following specific design patterns and techniques. Unfortunately, when applied to network packet processing tasks, these techniques limit code reuse and modularity, requiring developers to use deprecated programming conventions.**

We propose a methodology for developing high-speed networking applications using Vivado HLS for C++, focusing on reusability, code simplicity, and overall performance. Following this methodology, we implement a class library (`ntl`) with several building blocks that can be used in a wide spectrum of networking applications. We evaluate the methodology by implementing two applications: a UDP stateless firewall and a key-value store cache designed for FPGA-based SmartNICs, both processing packets at 40Gbps line-rate.

I. Introduction

Writing reusable and modular code is an important programming principle because it shortens development times, simplifies testing, and reduces bugs. In FPGAs, the increasing adoption of high-level synthesis (HLS) promises more opportunities for reuse than with traditional hardware definition languages because HLS tools abstract low-level hardware details. These tools allow developers to adapt code and extend functionality on different FPGAs and even target different performance goals. The tools achieve this through automatic pipelining and by providing internal scheduling. With the rapidly rising deployment of FPGAs in datacenters [1]–[5], increasing developer productivity is an important goal.

Network packet processing is one important domain that can benefit from HLS. Packet processing applications are in critical need of high-performance FPGA-based accelerators, due to rising network speeds and stagnating CPU performance. I/O intensive network functions such as load balancers, firewalls, and cryptographic gateways can benefit from hardware offloading and are a promising target for HLS [6]–[11].

In practice, however, optimizations and heuristics for HLS tools may require designers to write code and organize modules in ways that hinder encapsulation and reusability for networking applications. This limitation stems, in the case of Xilinx Vivado HLS [12] (C++ version), from its traditional target use-cases: computationally intensive algorithms supporting a restricted C++ dialect, focusing on, e.g., image processing, scientific computations, and machine learning. As other, similar tools, it supports basic data structures such as arrays and queues, which use simple methods that can be inlined, but it lacks support for more complex data structures [13]. Unfortunately, packet processing applications typically use advanced data structures such as hash tables, CAMs, and priority queues. Furthermore, current packet processing designs in Vivado HLS follow a dataflow programming methodology [14] to expose pipeline and task parallelism and use low latency streaming interfaces. Existing application examples written in Vivado HLS [7], [14], [15] use static variables for keeping state, and use functions rather than classes, thus severely limiting the ability to reuse modules and data structures across projects.

In this work we set out to increase code reusability for packet processing applications written in Xilinx Vivado HLS. We present our proposed methodology and design patterns that enable code reuse. We use modern (post C++11) techniques with object-oriented and template programming to write modular and generic code, while still enabling HLS optimizations to generate line-rate capable designs. Our methodology allows creating reusable building blocks that can be quickly assembled into complex applications, while optimizing the resulting design for high throughput and low latency (Section III). We describe several building blocks common in packet processing applications, such as header manipulations, hash tables, and schedulers (Section IV). We implement these building blocks as part of a new library for Vivado HLS called `ntl`[1]. Even though we target Vivado HLS for C++, the ideas in this work are applicable to other HLS tools as well.

We evaluate our proposal and the `ntl` library by implementing two networking applications: a key-value store cache and a UDP-based firewall for FPGA-based SmartNICs, showing that our methodology can simplify the implementation of high-performance networking applications using HLS. When comparing modules targeting the same line-rate behavior written using `ntl` against a high-performance framework for FPGA network processing (P4/SDNet), `ntl` improves latency by $8.4\times$ and reduces area by $6.5 - 16.1\times$ (Section V).

II. Background: Vivado HLS dataflow optimization

Many packet processing applications can be modeled as a dataflow graph [16], passing packets between processing

This research was supported by the Israel Science Foundation (Grant No. 1027/18) and by the Israeli Innovation Authority consortium.

[1]Networking Template Library: https://github.com/acsl-technion/ntl

2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00036

elements along a directed graph. This model maps well to hardware and is expressible in Vivado HLS, which offers specific optimizations that extract a dataflow graph from function definitions, synthesizing processing elements for function invocations and connecting them with FIFO buffers.

The Xilinx application note for protocol processing systems [14] instructs developers to create pipelined designs using the dataflow optimization, building elements as functions, and composing elements by calling them within other functions. Rather than executing complete tasks, functions are expected to take a small step toward their goal and return immediately, expecting to be called repeatedly. This allows the compiler to optimize them for high throughput.

State and internal interfaces are implemented using static variables. For example, a function may implement a state machine element with the current state as a static variable. Each invocation calculates the next state and updates the variable, returning immediately. This model matches the non-HLS semantics of C and C++, i.e., as software: a static variable keeps its state between function invocations.

Unfortunately, Vivado HLS's dataflow optimization has several limitations [12]: it forbids bypassing elements or creating feedback loops and requires each variable to have a single producer and consumer. Defining the I/O FIFOs of each element explicitly through `hls::stream` objects and disabling strict checking through a compiler flag allows implementing feedback loops, but the requirement to express state as static variables prevents element reuse across modules.

For example, under the dataflow optimization, an element may be reused by invoking its function more than once. However, if the function uses static variables, multiple instantiations share these variables, breaking the dataflow optimization. One may still create multiple instances of a given HLS function after compiling it to RTL. However, this approach requires adding glue logic that could introduce bugs, complicate the development process, and hinder code reuse in other projects.

```
class map {
public:
  template <typename InputStream, typename OutputStream,
            typename Func>
  void step(InputStream& in, OutputStream& out, Func&& f)
  {
#pragma HLS pipeline
    if (in.empty() || out.full()) return;
    out.write(f(in.read()));
  }
};
```

(a) Map higher order function definition.

```
template <typename In, typename Out, bool out_every_flit>
class fold {
public:
  stream<Out> out;

  explicit fold(const Out& initial) :
    _initial(initial), _current(initial) {}

  template <typename Func>
  void step(stream<In>& in, Func&& f) {
#pragma HLS inline region
    if (in.empty() || out.full()) return;

    auto flit = in.read();
    auto next = f(_current, flit);
    if (out_every_flit || last(flit)) out.write(next);
    _current = last(flit) ? _initial : next;
  }
private:
  const Out _initial; Out _current;
};
```

(b) Fold definition suitable for packet processing.

```
template <typename T, typename Counter = ap_uint<16> >
class counter : public fold<T, Counter> {
public:
    typedef fold<T, Counter, true> base;
    counter() : base(-1) {}

    void step(typename base::in_t& in)
    {
#pragma HLS pipeline
        base::step(in, [](const Counter& cnt, const T& t) {
            return Counter(cnt + 1);
        });
    }
};
```

(c) Count the flits in each packet using fold.

Fig. 1. Dataflow element examples using higher-order functions.

III. Design

A. Design considerations for packet processing

Our goal is to form a methodology for creating reusable packet processing components. Such applications are commonly implemented as *dataflow graphs*, passing packets and headers between concurrent processing units to exploit inter-packet parallelism and achieve high throughput [6], [7], [14].

Packets of various sizes are typically passed using fixed-width buses over multiple cycles, matching an external link bandwidth. Consequently, processing units typically have to keep state, even when packets are independent.

Packet processing applications are commonly split into *data plane* and *control plane* parts. The data plane aims for line-rate processing and low latency, while the control plane handles configuration and management tasks that are not on the performance critical path.

B. Design Methodology

To create reusable code, we build packet processing applications as a dataflow graph of reusable C++ classes, and use higher-order functions to customize them. These classes can further be composed through aggregation.

Dataflow element pattern: Using the dataflow optimization requirements, we define a pattern for expressing processing elements. Each basic element is defined as a C++ class, with a `step` method that synthesizes into the basic element's hardware counterpart. The `step` function only uses `hls::stream` arguments or `hls::stream` member variables for I/O.

Classes keep state as member variables, and only the class at the top of the hierarchy needs to be instantiated as a static variable of HLS's top function. This allows code reuse while also enabling Vivado HLS to perform dataflow optimizations.

A class may include methods other than `step`, but they

must be inlined into their calling function. These methods are forbidden from accessing the class state by the Vivado HLS dataflow optimization requirements. They can only use `hls::stream` member variables to communicate with the `step` function. We use such inline methods to simplify the API of more complex elements (e.g., the hash table in Section IV-C).

Higher-order functions: Many data manipulation operations can be described as different applications of higher-order functions such as *map* or *fold* over the same generic interfaces [17]. We apply the same principle to packet streams and show two examples of such processing elements.

A *map* element (Figure 1a) receives a sequence of inputs from an input stream and applies a given function to all input items, writing the results to an output stream. A *fold* element (Figure 1b) computes a scalar value from a stream. It is stateful, updating its internal state using a provided function.

When handling packet streams rather than scalar values in streams, each packet is composed of multiple flits. A common pattern we implement in our projects maintains a per-packet state, resetting it after each packet and updating the *fold* output once per logical packet rather than per physical input word.

Figure 1c shows an example counter unit, which uses fold. The counter's `step` method includes one explicit port: an input stream to count. In addition, the counter exposes an output port as the member variable `out`. The class generates a stream of counter values and not a scalar value to allow its use in dataflow optimized caller functions.

Dataflow subgraph pattern: More complex elements can be built using several basic ones. Our design pattern for this case instantiates the latter as private member variables and calls their `step` methods from the `step` method of the containing element. The container `step` method is inlined into its caller to facilitate the dataflow optimization, as suggested in [14]. Streams connecting the internal components are instantiated as additional private members of the composite class.

Figure 2 shows an enumerate element, which associates each input flit with a running counter. Designers may integrate multiple instances of such elements into larger dataflow graphs.

We envision a rich combinator library that makes it easy to reuse existing elements and to describe complex dataflow graphs as a series of combinator applications, as done, for instance, in reactive programs [18], [19].

C. Control plane

Packet processing applications use their control plane to configure high-level behavior and for monitoring purposes. It encompasses operations such as configuring flow tables, reading statistics registers, and accessing debugging information. The control plane does not need to process at line rate, and can be managed, for example, from a CPU via an AXI4-Lite interface. Vivado HLS can generate such interfaces from a top function's input and output arguments. In many cases, the control plane hardware must implement some transactional interface, to configure lookup table entries, for example. However the generated AXI4-Lite interface cannot easily detect a write transaction.

```
template <typename T, typename Counter = ap_uint<16> >
class enumerate {
public:
  typedef std::tuple<Counter, T> tuple_t;
  stream<tuple_t> out;
  void step(stream<T>& in) {
#pragma HLS inline
    dup.step(in);
    _counter.step(dup._streams[0]);
    zip.step(_counter.out, dup._streams[1]);
    link(zip.out, out);
  }
private:
  dup<T, 2> dup;
  counter<T, Counter> _counter;
  zip<tuple_t, Counter, T> zip;
};
```

(a) An enumeration unit that associates an increasing number with each flit in a packet, composed of multiple elements.

(b) Enumeration element block diagram.

Fig. 2. Enumeration example, composing multiple elements.

```
template <typename T> struct gateway_registers {
  ap_uint<31> opcode; ap_uint<1> go;
  T data;             ap_uint<1> done;
};

template <typename T> class gateway {
public:
  template <typename Func>
  void step(gateway_registers<T>& r, Func&& f) {
    if (r.go && !axilite_gateway_done) {
      if (f(r.opcode, r.data))
        axilite_gateway_done = true;
        r.done = 1;
      }
    } else if (!r.go && axilite_gateway_done) {
      axilite_gateway_done = false;
      r.done = 0;
    }
  }

private:
  bool axilite_gateway_done;
};
```

Fig. 3. Control plane gateway protocol implementation. The `r` parameter is exposed through AXI4-Lite.

To implement a higher level transactional interface on top of an AXI4-Lite interface, we design a simple gateway interface (Figure 3). The gateway includes registers for the chosen opcode and associated data (parameterized by the data type `T`), as well as "go" and "done" bits to control and expose the transaction state. The gateway `step` function is parameterized with a callable type `Func`, which it invokes for new transactions. The function may return `false` to indicate it has not finished the transaction, requiring more cycles for completion. In such cases it is up to the callable to maintain its state between calls, e.g., using a closure that captures references to its necessary state.

With a large design, this approach can simplify the resulting AXI4-Lite unit. Vivado HLS generates a single decoding unit containing all the registers exposed through the bus and connecting it to any unit in the design with control registers. By

TABLE I
MAIN BUILDING BLOCKS OF NTL.

Name	Description
`map<In, Out, Func>`	Apply `Func` on each input flit.
`fold<In, Out, Func>`	Aggregate the input stream using `Func`.
`dup<T, N>`	Duplicate a stream to N outputs.
`zip<Out, In1...>`	Combine a flit of each input stream.
`link<In, Out>`	Connects two streams.
`stream<T, Tag>`	Specialized stream interface.
`pack_stream<T>`	Stream with automatic (de)serialization.
`pfifo<T, Depth>`	Stream with a full-threshold.
`pop_header<Bytes>`	Pop a fixed-width header.
`push_header<Bytes>`	Push a fixed-width header.
`push_suffix<Bytes>`	Push a fixed-width suffix.
`array<T, Size>`	BRAM array w. control-plane interface.
`hash_table<K, V, Size>`	BRAM hash-table data structure.
`scheduler<N>`	DRR scheduler with N entries.
`gateway<T, Func>`	Transactional AXI4-Lite interface.

using a single gateway interface to multiplex several different transactions, we simplify the decoding unit and distribute it among the various elements, reducing routing congestion.

D. Integration with Vivado HLS

The patterns described in the previous sections make code reuse and encapsulation in Vivado HLS possible. We rely on C++ features to express general designs with template parameters that can be later set to the specific project requirements. Parameters such as bus widths, data types, or even functions and algorithms can be used. As tools resolve these parameters at compile-time, they are well suited to Vivado HLS requirements, as opposed to dynamic approaches such as polymorphic classes. However, some forms of generalizations are not well-supported in Vivado HLS. For example, using an array of processing elements can sometimes trigger dataflow violations due to multiple methods accessing the same member variable, even though each method is accessing a different array cell. We work around this limitation using the Boost preprocessing library [20], but this can make code reuse more difficult.

Vivado HLS's dataflow optimization typically uses a strict interpretation of its single-producer-single-consumer rule that can sometimes be overly limiting. For instance, when synthesizing processing elements from C++ methods, Vivado HLS interprets different method invocations of a single object as a violation of this rule, even when each method uses different member variables. We trigger these rule violations when implementing inlined accessor methods to our elements. Luckily, a compiler flag can disable this strict interpretation of dataflow rules, as a workaround. We expect that improved compiler analysis will eliminate such false violations in the future.

IV. LIBRARY BUILDING BLOCKS

While implementing networking applications in our group, several building blocks emerged, which we collected into the `ntl` library (Table I). The methodology described above allows implementing templated versions of these blocks that can be used in different applications and instantiated in different versions inside a single application.

A. Specialized streams

Vivado HLS uses the `hls::stream` template class to describe various FIFOs and streaming interfaces. However, the semantics of a `hls::stream` can vary depending on the underlying FIFO or interface it requires, and, in some cases, the developer may need a more complex FIFO abstraction.

Depending on its chosen implementation, `hls::stream` provides both blocking and non-blocking operations. Blocking operations may stall the element's state machine until the stream is ready to provide an output or accept an input, whereas non-blocking operations are asynchronous. Non-blocking writes are normally needed with FIFOs to prevent a deadlock, as a blocking write may cause the element to stop processing completely, including processing that could free space for the write to proceed. However, `hls::stream` objects that represent an AXI4-Stream output interface forbid non-blocking write operations, as these would create a dependency between the AXI4-Stream interface's TVALID output and its TREADY input, violating the AXI4 specifications.

We identified several alternative stream classes that cover the requirements of most packet processing applications (for instance, a programmable threshold FIFO, explained below). Users of these classes can use type polymorphism to write generic code that works with any of them (see Figure 1a).

Furthermore, to prevent mistakes, we wrap `hls::stream` with specialized wrappers according to its use. All wrappers implement the same stream concepts: an input stream has a read and empty methods, and an output stream has a write and full methods. The wrappers have two template type parameters. The first is the type of the values the stream contains, and the second is a tag that selects the read/write methods using template specialization: an ap_fifo interface, AXI4-Stream input, or AXI4-Stream output. Code that uses the specialized stream wrappers can work with the different interfaces transparently, as the underlying implementation chooses blocking or non-blocking operations according to the chosen interface.

Using the same interface, more complex stream classes can be defined. For example, we implement a `pack_stream` template that automatically packs and unpacks values to a raw bitstring representation. Even though Vivado HLS provides a `data_pack` directive that can be used to pack struct fields automatically, it leaves the decision of field ordering and padding to the compiler, with limited user control. Using the packing stream template, we can let developers write their own serialization and deserialization functions.

Example: Programmable threshold FIFO: Vivado HLS can automatically pipeline complex computations in processing elements. However, dependencies between pipeline stages can reduce the throughput. A common dependency in our methodology occurs between a check that an output stream is full and the actual write to that stream. If the compiler cannot schedule the two operations in the same cycle, it will increase the pipeline's initiation interval to the number of cycles between the two, resulting in artificially lower performance.

Our solution in the `ntl` library is to provide an alternative stream class that replaces the stream check with a credit-based

Fig. 4. An HLS FIFO with a programmable threshold. The custom logic and state are inlined into the producer and consumer modules. The producer keeps a local copy of the consumer index to enable a local credit check.

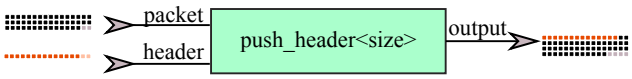


Fig. 5. The `push_header` element receives two stream interface inputs for the packet payload and the new header. It reorders the output to place the header before the payload in the output stream. The template class is parameterized by the header size to simplify the resulting RTL code.

mechanism. The programmable FIFO uses credit registers both at the producer and the consumer side, counting the number of elements they have seen (Figure 4). It uses a separate stream to pass credit updates between the two. Its modified full method checks whether the number of credits at the producer is below the given threshold, and the write method updates the producer's credits.

As the new full method does not access the underlying data FIFO, the compiler does not infer a dependency between fullness checks and writes. Thus, we replace the compile time dependency check of Vivado HLS with a runtime check, eliminating the unwanted dependency and improving throughput. When correctly configured, this design does not allow overflows.

Timely processing of credit updates from the consumer is necessary for the correct operation of the programmable FIFO, so updates cannot be restricted to the same conditions the FIFO user may check before calling the write or full methods. Therefore, we require the producer to call a `step` function of the programmable FIFO to read and process the credit updates on every invocation of the producer unit.

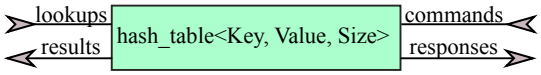
B. Header manipulation elements

Packet interfaces transmit data as a stream of fixed width flits, possibly fragmenting network headers across several flits. We design reusable elements to push/pop fixed-size headers to/from a packet stream.

Popping a header involves buffering the header octets and reordering the output payload to align it to flit width. The unit also sends the stripped header on a separate output stream, to allow independent processing of the header and the payload. Similarly, pushing a header requires buffering octets to realign the input payload while accommodating the header (Figure 5). A similar element pushes a fixed size suffix at the end of the packet, shifting it to align with the packet's current length.

C. Random access data structures

We implement two generic BRAM-based tables for data plane processing with slightly different semantics: an array and a hash table. Both allow lookups and updates at a high rate,



(a) Hash table element interfaces.

```
template <typename Key, typename Value, unsigned Size>
class hash_table {
public:
  /* Lookup interface */
  stream<Key> lookups;
  /* Lookup responses */
  stream<optional<Value> > results;
  void step();
  /* Inlined accessor methods */
  int add_entry(const Key& key, const Value& value,
                bool& result);
  int delete_entry(const Key& key, bool& result);
private:
  /* Updates interface values (see below) */
  typedef command_template<Key, Value> command;
  /* Helper method for implementing updates */
  int execute_command(const command& cmd, bool& resp);
  bool command_sent;
  /* The hash table */
  hash<Key, Value, Size> _table;
  /* Internal update interface streams */
  pack_stream<command> commands;
  pack_stream<bool> responses;
};
template <typename Key, typename Value>
struct command_template {
  enum command_enum { HASH_ADD, HASH_DELETE } cmd;
  Key key; Value value;
};
```

(b) Hash table class interface.

Fig. 6. Hash table element.

and both offer a control plane interface for setup (designed using the methodology described in the previous section).

The hash table class (Figure IV-C) provides a data plane interface for lookups and matching results and a control interface for updates. The lookup/result interface follows the dataflow element pattern, and the class exposes inline methods for callers to update or erase elements (`add/delete_entry`). These methods send formatted commands to the class's internal `commands` stream. The unit's `step` function reads the commands and writes a response to the `responses` stream. As this is an asynchronous process, the accessor functions may return a busy indication and be called again later to test that the response has arrived. The `command_sent` member variable tracks the state of this interface.

As an optimization to save logic resources on the device, arrays expose their internal data structure to the user without a stream, through an inline `operator[]` method (subscript operator). To comply with Vivado HLS dataflow optimization rules, only a single unit may call the method. This makes it difficult to implement control plane accesses to the array. We solve this issue by sending commands from the control plane gateway to the element that contains the array. The element calls another inline method of the array that handles incoming commands from the gateway.

D. Customizable scheduler element

Several applications we examined used a scheduler to allocate resources such as network bandwidth or computational power.

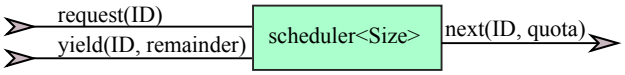


Fig. 7. A DRR scheduler element. The scheduler has two input streams: requests for scheduling and yield commands. It outputs a stream of the next scheduled task. We omit the Control plane interface for programming quotas.

We design a scheduler (Figure 7) as an abstract building block well suited for both use cases. Our scheduler implements the deficit round robin (DRR) algorithm [21], but in the future it could be easily extended with other algorithms as well.

The scheduler receives requests to schedule different entities (flows, threads, etc.) and enqueues them. It allocates a quantum (e.g., a number of bytes to transmit, or a timeslice) to an entity and sends its allocation to an execution unit. When the execution unit finishes the quantum, or there is no more work for the current entity, the execution unit notifies the scheduler while reporting the remainder of the quantum, and reads the next entity and its quantum.

We implement the scheduler as a dataflow element, with a gateway instance for programming its quotas and other configuration parameters from the control plane.

V. Evaluation

Applications such as network address translation (NAT), load balancing, or stateful firewall, can use several `ntl` elements: hash-table elements for per-flow state, gateways for configuration, header manipulation elements for (de)parsing packets, and map and fold elements for glueing other elements together. We evaluate the efficiency of our proposed HLS development methodology and library using a firewall and a key-value cache.

As our target platform we use a Mellanox Innova Flex card [22] which has a Xilinx Kintex UltraScale XCKU060 FPGA (xcku060-ffva1156-2-i) and is attached to a host via PCIe. We use Xilinx Vivado HLS 2018.2. Our code and methodology, however, is easily portable to similar SmartNICs as it relies on Vivado HLS for everything but the vendor specific code and DDR memory management unit (MMU). The vendor provided FPGA shell dictates a specific clock rate (216.25 MHz), so we target this clock rate in both applications.

A. UDP firewall

Our first example is a UDP firewall that receives a stream of packets, parses their headers to find IP and UDP fields, and uses a hash-table to classify the packets based on their source and destination IP addresses and UDP ports. Data is received over the card's 40 Gbps network port, and filtered packets are sent to the host CPU over PCIe.

The `ntl`-based HLS implementation uses a *duplication* building block to buffer incoming packets before processing them. Its parser is based on the *enumeration* element (Figure 2), and a *fold* instance (Section III-B). The *fold* instance uses the flit count from the enumerator together with the provided flit to extract the necessary header fields. The rest of the firewall pipeline uses a *map* instance to extract the hash key from each packet header, and a *hash table* instance. We use another `ntl` building block (*zip with*) to merge streams that influence the forwarding decision and apply a lambda function on their values. In addition, the firewall uses a control plane gateway for the UDP port table configuration.

TABLE II
Firewall application performance, area, and lines of code.

	II	Latency	LUTs	FFs	BRAM	LoC
HLS	3	25 cycles	5296	7179	12	218
HLS legacy	3	16 cycles	4087	4287	12	593
P4	2	211 cycles	34531	49042	193	92

We compare the `ntl`-based HLS implementation against a legacy HLS implementation which contains hand-written modules (only few of these reusable) and follows the recommended coding standards from Xilinx. Furthermore, we compare to a P4 implementation compiled using Xilinx SDNet 2018.2. The legacy HLS implementation has a functionally equivalent design. We develop it based on the `ntl` version, but refactoring it to compile with C++98 rather than C++11 and using static variables to describe stateful elements. This results in many fewer reusable elements than in the library version of the code. The P4 implementation describes an identical parser and flow table and uses the same conditions in its control flow.

Table II compares code complexity across variants by counting the lines of code using the cloc tool [23]. The `ntl`-based application is $2.7\times$ shorter than the legacy HLS implementation due to two reasons: first, the `ntl` library (1098 LoC) contains elements that the legacy implementation must customize and duplicate (though we could use the `ntl` elements as starting points). Second, by using higher-order functions, the `ntl` version allows writing more succinct code even when the application requires custom functionality.

The HLS implementation requires only twice as many lines compared to the P4 implementation, not counting the `ntl` library code that the developer does not need to change (see Table II). This difference is not surprising because P4 is a domain specific language and it provides simpler syntax for building parsers and connecting elements. However, a general-purpose tool such as HLS allows fine-tuning the design for better performance and, as we show next, implementing algorithms that are not expressible in P4.

We compare the performance of the resulting FPGA circuits in Table II. It shows that all implementations can process 64-byte packets with 3 cycles between packets (P4 achieved higher rate than targeted), providing throughput of 72 million packets per second (Mpps), enough to sustain line rate (59.5 Mpps). However, the `ntl`-based implementation delivers $8.4\times$ lower latency than the P4 version while also saving resources ($6.5\times$ lower for LUTs and $16.1\times$ lower for BRAMs). The legacy HLS implementation achieves the same throughput and somewhat lower latency than the library-based version. Its area use is smaller, but both HLS versions use an order of magnitude less resources than the P4 version.

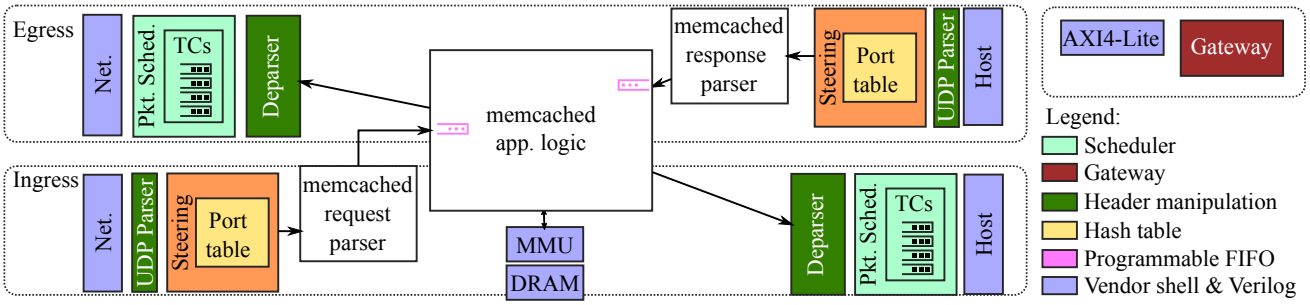


Fig. 8. Key-value store in-NIC cache block diagram. Top: egress pipeline handling packets sent from the host towards the network. Bottom: ingress pipeline for packets received from the network. Reusable elements are color coded.

B. Key-value store cache

To show our framework with a more complex application, we develop a prototype of a multitenant key-value store cache. We augment memcached [24] with a transparent write-through cache on the SmartNIC (similarly to previous work [25], [26])[2]. The design exposes the accelerator services to software using the NICA framework [27].

The accelerator on the FPGA parses incoming GET requests encoded using UDP packets and responds directly to the client for cache hits, while forwarding cache misses to the host CPU. The cache is filled transparently by snooping on the host CPU GET responses on the TX path. To keep the cache coherent with the host, incoming SET requests invalidate their respective cache entry (if present).

Figure 8 shows the block diagram of the key-value store cache. Network packets are first classified to determine whether they belong to the accelerator. Unrelated packets are forwarded directly to the host. The classifier uses a similar design to the UDP firewall, utilizing the *hash table* template class. In addition, the UDP parser and de-parser use the *header manipulation* classes to split header and data to different streams, which allows them to be processed at different rates.

Ingress packets targeting memcached are then processed with an application-layer parser and generate DRAM read commands for GET requests and DRAM write commands for SET requests. Some operations, especially parsing and hashing, require deep pipelines to meet the necessary throughput, and utilize the *programmable FIFO* building blocks.

We generate memory access commands using a stream interface that is bound in a Verilog wrapper to an AXI4-MM interface of the DDR controller. We chose not to use Vivado HLS's automatic AXI4-MM generated interface because it could only generate a limited number of outstanding requests, limiting the application performance for small requests. DRAM responses are processed to decide whether the access is a hit or a miss and handled accordingly. Hits generate a response that is sent out over the network, while misses cause a buffered copy of the request to be passed to the host. SET requests are always passed to the host. The DRAM cache structure is a hash table that uses the keys as hashes and stores the values inline. We support keys and values of up to 16 bytes.

[2]Here we do not compare `ntl` and P4 as P4 expresses all operations as packet manipulations and is insufficient for implementing the cache.

TABLE III
KEY-VALUE STORE CACHE PERFORMANCE AND AREA

Module	LUTs	FFs	BRAM	HLS LoC	RTL LoC
Infrastructure	44347	57448	565	6643	1736
Key-value cache	15383	15256	73	975	0
Vendor shell	170179	213942	309	-	-

Our design supports multiple tenants, each associated with a traffic class; an administrator can prioritize the different classes from software. The UDP parser tags requests and responses with a tenant ID, and each tenant has a separate memory area for storing key-value pairs. We instantiate two elements of the *DDR scheduler* class, one for the network interface and one for the host interface. Furthermore, each tenant has separate configuration and statistics registers; these are implemented using an instance of the *array* template class (a gateway interface provides software access).

The main building blocks required to provide arbitration, buffering and packet reassembly for multiple tenants are very similar to those in related work on multi-tenant key-value stores written in RTL, for instance Multes [28]. For this reason, using HLS-based modules, such as the ones in `ntl`, will be beneficial to emerging cloud-based FPGA designs that offer network-facing multi-tenant services.

We synthesize the accelerator for 64 tenants, a table with 1024 entries, and 4 traffic classes. As Table III shows, the resource requirements are modest. More importantly, thanks to the `ntl` library, the number of lines of code necessary for expressing the key-value cache is small (similar order of magnitude to the UDP firewall). This shows reduced development effort. In terms of performance, the accelerator processes GET requests with 16-byte keys and values at line rate (40.3 Mtps), showing that our design methodology results in code that HLS tools can optimize and that yields high throughput designs.

VI. Related Work

The Xilinx methodology for packet processing [14] has been successfully applied in projects such as a TCP/IP stack [7] and memcached server [6], [15]. This methodology uses a

dataflow design, but its use of static variables for holding state severely limits code reusability. In this work we extend this methodology for better code reuse by using C++ classes to wrap processing units, allowing multiple instantiations of the same unit.

In [13] the authors present an architectural template for using complex data structures in HLS. Each data structure unit is composed of multiple specialized method units (SMUs), a dispatcher unit, and a collector unit. Some of our data structure building blocks, e.g., the array and the hash table, follow a similar pattern, although we use custom dispatchers and collectors to save resources. Unlike [13], our methodology allows the pipeline and all building blocks to be expressed in HLS, resulting in less RTL glue logic overall.

Silva et al. [29] present an HLS design methodology for high performance, low area, and code modularity using modern C++ features, with example packet processing applications, among others. We share several aspects of the methodology, but we focus on packet processing requirements such as dataflow, and provide a class library for networking elements.

Emu [10] is a framework for hardware networking applications that uses high-level synthesis from C# with the Kiwi [30] HLS tool. Kiwi supports a different dataflow model than Vivado HLS, using concurrent threads to describe the different processing units, so it uses different design patterns. Nonetheless, we share Emu's goal of implementing a library for common networking elements with HLS.

Researchers have developed domain-specific languages (DSLs), such as P4 [31] or ClickNP [8], intended for packet processing and networking applications running on specific hardware devices, including FPGAs [9], [11], [32]. These restrict the capabilities of the packet processing steps in the pipeline, especially when it comes to expressing complex data structures, to ensure that the resulting behavior is mappable to the underlying programmable networking hardware. Conversely, our work uses a general-purpose HLS tool, allowing users to write packet processing applications with rich functionality.

Several projects separate the definition of the dataflow graph and its processing elements. Floem [25], for instance, compiles a DSL for CPU-based SmartNIC programming, combining a Python-based dataflow model and elements programmed in C. Similarly, Maxeler MaxJ [33] provides an HLS platform for dataflow programming, using an extended Java language variant. MaxJ divides programs into one or more kernels and a manager that links them, rather than aggregating kernels into higher level kernels. Unlike the above, we use C++ and HLS as a single language and a single abstraction to develop both the dataflow graph and the individual elements.

Vivado HLS includes an image and video processing library based on OpenCV [12]. This library is designed for dataflow processing of images, passing images as streams for a pipelined operation, and providing building blocks such as filters and transformations. Even though it targets a different application domain than our work, both use dataflow optimized designs. However, OpenCV HLS library functions do not keep state between invocations, so their building blocks are functions rather than classes. In addition, they do not use higher-order functions to customize generic algorithms or patterns.

Previous works used higher order C++ functions for HLS [17], implementing parallel dataflow algorithms focusing on HPC designs [34], image processing [35], and using C++ meta-programming to provide recursion [36]. We apply similar techniques for packet processing.

VII. Discussion and Future Work

Developing a large and feature-rich application, such as the key-value cache, has allowed us to better understand and enhance our methodology. Some patterns have appeared several times in the design, which demonstrated that they were good candidates for the `ntl` library. These patterns appear across many networking applications. For example, both the generic flow classifiers and our application-specific accelerators buffer packets until their fate is decided, and we use a common pattern for that. In addition, the designs share common parts in both their RX and TX pipelines (ingress and egress). Implementing these as a shared class, instantiated per pipeline, accelerates development and reduces errors.

We found that it was not feasible to build our example applications with HLS alone. We had to use Verilog and external IP for things like the MMU and interaction with the Vendor provided shell. We designed the MMU as a separate unit to simplify the application design, allowing it to use virtual addresses. Vivado HLS can generate AXI4-MM masters, but not slaves, so we chose to implement the MMU in Verilog. Since we have changed the HLS design to use AXI4-Stream interface to DDR, in future projects it will be possible to implement the MMU in HLS and use Verilog only for the glue logic connecting the HLS interfaces with the DDR.

Looking forward, we envision translating our methodology to toolchains beyond Vivado HLS. For instance, SYCL [37] is a standard for heterogeneous development with modern C++, based on OpenCL, with implementations for FPGAs under development [38]. We share SYCL's goal of using modern C++ for FPGA development, and the desire for single-source compilation of heterogeneous applications. SYCL may provide the same building blocks we use for dataflow programming, with its pipes abstraction. Our prototyping platform (a Mellanox Innova Flex card) does not currently support OpenCL, so we implement our library over Vivado HLS and leave a SYCL-based implementation for future work.

VIII. Conclusion

High-level synthesis promises faster development for FPGA designs but often suffers from poor code reusability due to the requirements for optimizations such as dataflow pipelining. We show how to create reusable and customizable building blocks for the network packet processing domain while generating high-performance efficient FPGA circuits with HLS tools. We validate the usability of our methodology by implementing two 40Gbps applications. While they perform different tasks (key-value caching vs. firewall), these two applications share numerous `ntl` building blocks, simplifying development.

REFERENCES

[1] Amazon Web Services, "Amazon EC2 F1 instances," 2016, (Accessed: Jan. 2019). [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[2] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, ser. NSDI '18. Renton, WA: USENIX Association, 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[3] Alibaba Cloud, "Instance type families: f1, compute optimized type family with FPGA," (Accessed: Jan. 2019). [Online]. Available: https://www.alibabacloud.com/help/doc-detail/25378.htm#f1

[4] Huawei Cloud, "FPGA-accelerated cloud server," (Accessed: Jan. 2019). [Online]. Available: https://www.huaweicloud.com/en-us/product/fcs.html

[5] OVH Labs, "FPGA accelerators on public cloud," (Accessed: Jan. 2019). [Online]. Available: https://labs.ovh.com/fpga-accelerators-on-public-cloud

[6] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. San Jose, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Blott

[7] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware," in *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 36–43. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2015.12

[8] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934897

[9] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A rapid prototyping framework for P4," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 122–135.

[10] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman, "Emu: Rapid prototyping of networking services," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 459–471. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/sultana

[11] J. Santiago da Silva, F.-R. Boyer, and J. M. P. Langlois, "P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 147–152. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174270

[12] Xilinx, "Vivado design suite user guide, high-level synthesis," UG902 (v2018.2), 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf

[13] R. Zhao, G. Liu, S. Srinath, C. Batten, and Z. Zhang, "Improving high-level synthesis with decoupled data structure optimization," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 137:1–137:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898030

[14] J. H. Kimon Karras, "Designing protocol processing systems with Vivado high-level synthesis," Xilinx application note XAPP1209 (v1.0.1), 2014. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf

[15] Xilinx, "HLS implementation of memcached pipeline," Accessed: Jan. 2019, 2016. [Online]. Available: https://github.com/Xilinx/HLx_Examples/tree/master/Acceleration/memcached

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[17] D. Richmond, A. Althoff, and R. Kastner, "Synthesizable higher-order functions for C++," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2835–2844, Nov 2018.

[18] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2501654.2501666

[19] "ReactiveX," Accessed: Jul. 2018, 2018. [Online]. Available: http://reactivex.io/

[20] P. M. Vesa Karvonen, "The Boost library – preprocessor subset for C/C++," 1.67, 2018. [Online]. Available: https://www.boost.org/doc/libs/1_67_0/libs/preprocessor/doc/index.html

[21] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, Jun 1996.

[22] Mellanox Technologies, "Innova Flex 4 Lx EN adapter card product brief," https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova_Flex4_Lx_EN.pdf, 2017, (Accessed: Sep. 2018).

[23] A. Danial, "cloc–count lines of code," https://github.com/AlDanial/cloc, 2018, (Accessed: Sep. 2018).

[24] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[25] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A programming system for NIC-accelerated network applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 663–679. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[26] Y. Tokusashi and H. Matsutani, "A multilevel NOSQL cache design combining in-NIC and in-kernel caches," Aug. 2016, pp. 60–67.

[27] H. Eran, L. Zeno, G. Malka, and M. Silberstein, "NICA: OS support for near-data network application accelerators," in *MaRS'17*, 2017.

[28] Z. István, G. Alonso, and A. Singla, "Providing multi-tenant services with FPGAs: Case study on a key-value store," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 119–1195.

[29] J. Santiago da Silva, F.-R. Boyer, and J. M. P. Langlois, "Module-per-Object: a human-driven methodology for C++-based high-level synthesis design," in *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '19. Washington, DC, USA: IEEE Computer Society, 2019.

[30] S. Singh and D. J. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 3–12.

[31] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[32] Xilinx Inc., "SDNet development environment," 2018, (Accessed: Jan. 2019). [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdnet.html

[33] Maxeler Technologies, "Programming MPC systems – white paper," Jun. 2013. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf

[34] J. de Fine Licht, M. Blott, and T. Hoefler, "Designing scalable FPGA architectures using high-level synthesis," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 403–404. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178527

[35] M. A. Oezkan, O. Reiche, F. Hannig, and J. Teich, "A highly efficient and comprehensive image processing library for c++-based high-level synthesis," in *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, Sep. 2017, pp. 1–10.

[36] D. B. Thomas, "Synthesisable recursion for C++ HLS tools," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2016, pp. 91–98.

[37] Khronos OpenCL Working Group – SYCL subgroup, "SYCL provisional specification," Feb. 2016, version 2.2. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-2.2.pdf

[38] R. Keryell and L.-Y. Yu, "Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended abstract of technical presentation," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '18. New York, NY, USA: ACM, 2018, pp. 18:1–18:8. [Online]. Available: http://doi.acm.org/10.1145/3204919.3204937