Creating Custom Network Packet Processing Pipelines on HMC-Enabled FPGAs

Jehandad Khan and Peter Athanas
Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia
Email: {jehandad,athanas}@vt.edu

***Abstract*—A higher tier of network packet processing performance can be achieved by augmenting the agility that FPGAs offer with the sheer streaming throughput offered by a Hybrid Memory Cube (HMC). The notion of a programmable data plane specified in a domain-specific language enables the creation of custom protocol and packet operations. This paper presents an effort to map one such domain specific language, namely P4 (Protocol Independent Packet Processing) to an HMC-enabled FPGA platform by using HLS as the intermediate representation. The use of HLS affords productivity advantages as well as enabling a close correlation between the P4 code and the corresponding hardware units required to achieve the functionality. The resulting code leverages the parallel nature of the HMC to yield a packet pipeline capable of delivering 30 million packets per second (Mpps) using only a single HMC channel, which translates to 30 Gbps of data throughput for a Layer-3 router with an average of 128-byte long packets. Demonstrated here, by using 10 HMC user channels, the system is capable of achieving 300 Mpps (or 300 Gbps) of throughput.**

I. HMC as a Packet Lookup Memory

The Micron Hybrid Memory Cube consists of multiple memory die stacked together and interconnected using through silicon via (TSV) technology. The bottom layer of the stack consists of a controller that controls the transactions to the memory above. The hallmark of the HMC is its fast multiport throughput for random memory access. Unlike traditional memory interfaces, which rely on on-device caching hierarchies to exploit locality of access for performance, the HMC can deliver an order of magnitude higher random access performance without local caching. High random access throughput and the ability to make atomic operations make HMC well suited for performing packet look-ups in a networking context.

Network packet processing requires the maintenance of little state between packets, which makes it an embarrassingly parallel operation. While table sizes, even for millions of entries, take only 100s of MBs, the random access rate makes memory performance critical. Since there is little to no correlation between individual packets, memory access for flow look-up does not follow any pattern. This characteristic, coupled with the dependency between flow rules, makes it difficult to explore on-chip cache lines or to require complex logic to make the use of off-chip memory possible. The traditional solution to this problem is the use of dedicated TCAM devices, which are expensive and power hungry and, therefore, less than ideal. This makes the HMC a good alternative for increasing table scale by offering off-chip memory access without compromising on throughput. The parallel nature of the HMC fits nicely in this scenario, while the ability to perform atomic updates presents interesting possibilities for maintaining packet counts, flow hit counts, and other statistics. Multiple parallel interfaces exposed by the HMC controller may be mapped to either different mapping stages or different physical interfaces coming into the FPGA. Prevalent research in packet processing using FPGAs have been restricted to the use of on-chip memory due to the trade-off between table scale and performance. On-chip memory in such scenarios is already stressed due to the stringent requirements for buffers in store-and-forward architectures. These also have a bearing on the maximum frequency of the design. Moreover, this puts a hard limit on the maximum possible table size. While the latency of the HMC may become an issue in some applications, the same effect may be offset using either speculative issue in a packet pipeline or the use of intelligent caching mechanisms.

II. Architecture

This section summarizes the hardware/software infrastructure of the P4-based platform. The proposed compiler implementation leverages as much of the existing open-source code as possible by making use of the already available P4 front-end. This reduces redundancy of effort and accelerates development by making available concrete data structures representing the P4 pipeline. The P4 front-end forms the first step in the compilation of P4 to an FPGA target. The following sections give further details on the subsequent steps:

A. Parser/Deparser Generation

To achieve maximum throughput, it is necessary that all of the incoming packet data are parsed in the same cycle by deciding on all possible branches; thereby consuming all of the available bits. This, however, creates a number of complexities including but not limited to the requirement of look ahead (some bits in the current FLIT (FLow control digIT) decide on the operation of the next FLIT) as well as splitting of fields between FLITs. The proposed implementation deals with all of these cases by emitting a parsed representation of the packet in the minimum number of cycles. The parser is also able to handle packets of arbitrary length by storing away the unused part of the packet in an on-chip packet server, which is reunited with the packet at the de-parsing stage.

B. Matching Stages

The compiler iterates over all the matching stages described in the P4 program to determine the matching algorithm suitable for implementation.

1) Exact Match: A hash-based look-up engine uses the crc32 hash value to compute the location of the stored item in an on-chip array. The compiler back-end generates action logic for each possible action specified in the P4 program, that may be retrieved from the memory location and therefore acted upon.

TABLE I
PERFORMANCE METRICS PER HMC CHANNEL

Packet Length	`simple_router` (Mpps)		`simple_nat` (Mpps)	
(Bytes)	128 bits	256 bits	128 bits	256 bits
48	28			
64	25		26	
96	18	28	19	29
128	14	25	14	25
256	7	14	7	14
512		7		7
1024		4		4

The compiler also generates appropriate emit paths as described in the ingress/egress control graph.

2) LPM / Ternary Match: The ternary match is implemented by the compiler as the merging of a look-up and update stage. The look-up stage consists of an on-chip bloom filter, which computes the hash for the matching fields specified in the P4 code, performs a hash-based look-up for each mask into the Bloom filter to determine if there might be a hit off-chip or not. A priority decoder decides the highest priority matching mask. The winning mask and hash are used to compute the off-chip HMC address. Finally, a read request to the HMC controller interface is issued. The update stage waits for the response from the HMC controller interface. Once received, it operates on the received action and action data in a manner similar to the exact match stage. The packet, which has now been updated as a result of the action processing, is forwarded to the appropriate path.

3) Host Interface / Control Plane: Note that P4 requires the contents of the tables to be set externally. Therefore, all the on-chip table contents, as well as the Bloom filter are set using a host-to-FPGA interface that is generated automatically during the execution of the compiler. The compiler also generates a hardware abstraction header file for updating each matching stage, and the addresses for the on-chip control registers that control the behavior of the system.

C. HLS IP Generation

Once the individual matching/action stages have been created, TCL scripts are created to compile the generated HLS code into RTL modules. These modules are subsequently exported to the IP-XACT format for consumption and querying by later stages of the compiler. This stage has to precede the rest of the compiler steps since the control graph generator module requires interface information to connect different modules.

D. Control Graph

Once the HLS IP has been created, the P4 intermediate representation is converted to a directed graph. This directed graph is overlaid with RTL-relevant information, such as the AXI-Streaming interface required to connect different modules, and the required control and data registers. Next, the necessary FIFOs between the look-up stages as well as the HMC interface module are inserted as nodes in the graph. As a final step, the entire graph is serialized to a synthesizable Verilog module. This graph also contains the packet server and FIFOs at critical points where the compiler deems necessary.

E. HLS as the Intermediate Language for DSLs

It might be noted that the compiler framework uses HLS as the intermediate language. This choice affords many advantages at the cost of some performance and area penalty. The use of HLS as the intermediate language enables the creation of automatic interfaces and their associated handshake logic. While it is possible to automatically generate this logic in a compiler framework, it is much more convenient to have the same automatically and reliably generated. Since much of a packet processing performance revolves around memory access optimization that requires the replication of memory units to ensure the successful processing of a packet *each* clock cycle. This is easily enabled using HLS directives making the process almost automatic and painless. Other pertinent benefits include automatic pipelining of designs and scheduling of operations. The generated code is human readable to a large extent making it easier to understand and debug should it fail to operate as expected. Finally, the performance and latency estimates provided by the tools act as first order estimates for the achievable performance. Despite all the benefits afforded by HLS, it is not without its quirks. It took the authors several months to iterate on the design and, therefore, the backend design patterns to achieve competitive performance. Some constructs had to be implemented in RTL, such as the packet server, to improve performance and avoid dependency penalties in the synthesis of hardware. However, if the compiler had to be implemented in Verilog or some other RTL, the development time would have been greater.

III. RESULTS

The target platform consists of a 4 GB Micron Hybrid Memory Cube coupled to a Xilinx Kintex Ultrascale XCKU060-FFVA1156 device in the form of a Pico Computing AC-510 daughter card. The FPGA interfaces to the HMC use a pair of quarter-width HMC links, making it an effective single half-width link. It may be noted that the FPGA has only access to a quarter of the maximum HMC throughput, which supports two full-width links. The Micron / Pico Computing AC-510 daughter card is hosted on a Pico EX-700 backplane, which can host a maximum of six such daughter cards. The backplane provides PCIe-based interconnect between the FPGAs as well PCIe interconnect to the host. Since the host-to-device interface is throughput limited, for the purpose of testing, the packet pipeline is supplied packets from an on-chip packet generator.

To test the proposed system, two P4 programs from the online *p4lang* repository, namely `simple_router.p4` and `simple_nat.p4` were chosen. Table I outlines the achieved performance for these designs for different packet sizes and FLIT widths. It may be noted that performance for a single HMC is reported, while nine additional such channels are available on the device.

IV. CONCLUSION

This paper presents the efficacy of HMC as the ideal memory for assisting packet processing on FPGAs in general and packet look-up operations in particular. Another major contribution of this paper is to demonstrate the effectiveness of HLS as an intermediate language for packet processing. This approach bears fruit in terms of productivity, ease of debugging and automatic optimizations of code. The drawback, however, is the limited expression and abstraction of hardware, which may sometimes form a wall between the underlying hardware and the developer.

The authors intend to release a stable version of the code in the hope that it would be of benefit to the community as well as broaden the scope of the back-end to handle non-HMC targets.