Compiling High Throughput Network Processors

Maysam Lavasani
Electrical and Computer Engineering
The University of Texas at Austin
maysam@mail.utexas.edu

Larry Dennison
Lightwolf Technologies, Walpole, MA
larry@lightwolftech.com

Derek Chiou
Electrical and Computer Engineering
The University of Texas at Austin
derek@ece.utexas.edu

ABSTRACT

Gorilla is a methodology for generating FPGA-based solutions especially well suited for data parallel applications with fine grain irregularity. Irregularity simultaneously destroys performance and increases power consumption on many data parallel processors such as General Purpose Graphical Processor Units (GPGPUs). Gorilla achieves high performance and low power through the use of FPGA-tailored parallelization techniques and application-specific hardwired accelerators, processing engines, and communication mechanisms. Automatic compilation from a stylized C language and templates that define the hardware structure coupled with the intrinsic flexibility of FPGAs provide high performance, low power, and programmability.

Gorilla's capabilities are demonstrated through the generation of a family of core-router network processors processing up to 100Gbps (200MPPS for 64B packets) supporting any mix of IPv4, IPv6, and Multi-Protocol Label Switching (MPLS) packets on a single FPGA with off-chip IP lookup tables. A 40Gbps version of that network processor was run with an embedded test rig on a Xilinx Virtex-6 FPGA, verifying for performance and correctness. Its measured power consumption is comparable to full custom, commercial network processors. In addition, it is demonstrated how Gorilla can be used to generate merged virtual routers, saving FPGA resources.

Categories and Subject Descriptors: C.5.m [Computer System Implementation]:Miscellaneous

General Terms: Performance, Design, Experimentation

Keywords: FPGA, network processor, templates

1. INTRODUCTION

Specialized hardware can be significantly higher in performance and significantly lower in power than programmable processors [13]. Designing and implementing high quality specialized hardware requires an expert. Unfortunately, few application domain experts are experts in high performance hardware design and vice-versa.

Gorilla helps to bridge the gap between hardware experts and domain experts, providing each with the ability to focus on what they do best and automatically combining the efforts to generate highly efficient hardware. Gorilla is designed to maximize utilization of the most critical hardware resources in the system. In many systems, the most critical resources are the external pins of the chip. Critical resources are encapsulated in *accelerators* that perform core domain-specific functions and contain appropriate critical state. Domain experts define accelerator functionality and algorithms in a stylized C (Gorilla C). Domain experts also write application code that contains explicit calls to accelerators in Gorilla C.

Gorilla uses parameterized *templates*, which are generally defined by hardware experts, to encapsulate efficient hardware structures. Gorilla's compiler accepts a template along with template parameters that include arbitrary functions, state machines, and constants and generates synthesizable Verilog that defines customized hardware. A *canonical architecture* is a template designed to implement a specific class of applications.

The Gorilla compiler is used to combine application and accelerator code with the appropriate templates to generate specialized hardware that implements the application. Gorilla generates Verilog that is intended for FPGAs but could be used to generate ASICs. However, when Gorilla is used to design FPGAs, the dual benefits of high efficiency specialized hardware that competitive with application-specific processors and full programmability through the FPGA fabric are achieved. In fact, Gorilla on FPGAs is more general than application-specific processors.

A family of network processors that achieve similar performance and power to application-specific processors while running on an FPGA at 100MHz is generated to demonstrate the Gorilla methodology. One instance, targeting a single Xilinx Virtex-7 VHX870T, achieves 100Gbps (200 Million-Packets-Per-Second (MPPS)) in a single FPGA while processing any mix of IPv4, IPv6, and multi-labeled MPLS packets. To our best knowledge this is the highest throughput for a FPGA-based network processor generated from a high level specification. A 40Gbps instance was run on a single Xilinx Virtex-6 XC6VLX240T FPGA on an ML605 prototyping board.

The network processing template is far more general than simply network processing. By replacing accelerators, it is especially well suited for streaming applications and data

FPGA'12, February 22–24, 2012, Monterey, California, USA.

parallel applications whose performance and power scale poorly due to fine grain processing irregularity caused by data dependencies and irregular access to shared resources. The potential for reuse by factoring functionality from the template in the Gorilla style is significant.

The contributions of this paper are as follows:

- The Gorilla methodology that includes a programming model consisting of stylized sequential C calling accelerators written by a domain expert, the concept of highly parameterized templates written and assembled by a hardware expert, and a tool chain that automatically combines the two to generate hardware comparable in quality to hand-written hardware.
- A library of infrastructure templates including hierarchical arbitration, load balancing, reordering queue, and rate adaptation templates. Their parametric nature enables easy exploration of the design space to find the best configuration for a specific application.
- A case study of a network processor family, capable of processing any combination of IPv4, IPv6, and MPLS packets at up to 100Gbps at 100MHz, that has been synthesized, placed-and-routed, and verified for correctness in available FPGAs. An IPv4-only version running at 40Gbps (100MPPS) was run on a Xilinx ML605 prototyping board, measured for performance and power, and verified for correctness. All were generated using the Gorilla methodology.

2. PACKET PROCESSING

Because we use packet processing as our example, we describe it here. Internet routers process incoming packets to determine which output port they should be forwarded to. How a packet is processed is defined by the first part of the packet, known as the *header*. Packets can be of different types, each type requiring different processing steps. A single type of packet could have a variable number of *labels* that change the number of total processing steps. Packets can also be *encapsulated* in other packets, potentially requiring the router to look beyond the first header to complete the processing.

Figure 1 shows a simplified chart showing the processing steps of a router supporting IPv4/IPv6/MPLS. The grey rectangles are steps that require global and/or shared state accesses. The entry point in processing of a packet in this figure is *Dispatch* step. It checks the packet layer 2 protocol and jumps to the appropriate state for processing the protocol header (In figure 1 we are showing only one layer 2 protocol which is Ethernet.) *Ethernet* step detects the layer 3 protocol and also checks the integrity of Ethernet header. Processing IPv4 (one of the possible layer 3 protocols) consists of extracting the packet fields, checking the integrity of the fields, classifying the packet by looking up the source and destination addresses, and finally updating the packet header fields. In many router applications, the source and destination IP addresses together determine the output port, though our simplified code does not reflect that. The destination port for the packet is determined using lookup process and packet is forwarded to that port. Processing IPv6 is similar to IPv4 except that IPv6 has longer addresses. A MPLS header may contain multiple labels. The next action is determined using the result of the MPLS label lookup, whether

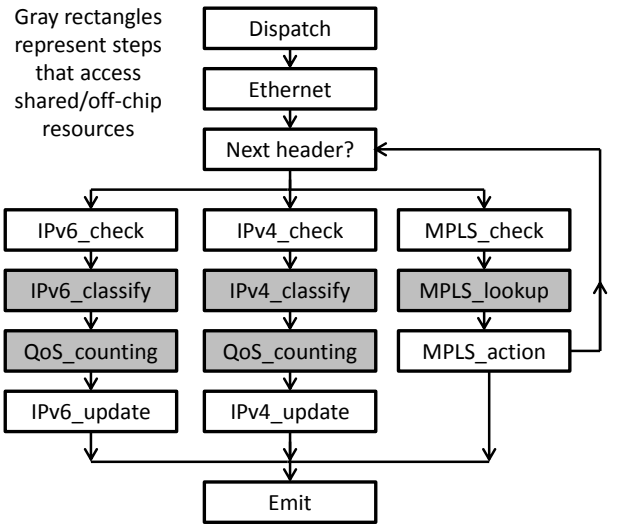

Figure 1: IPv4/IPv6/MPLS Packet Processing

it is processing another MPLS label, deleting the current label, attaching a new label, or processing another encapsulated protocol. The exit point for processing a packet in Figure 1 is the *Emit* step.

3. GORILLA METHODOLOGY

In this section we describe the Gorilla programming model, its tool chain, and the concept of canonical architectures.

3.1 Programming Model

Gorilla domain code consists of a sequence of steps, where each step is C code that (i) calls accelerators that implement specific functionality and handle accesses to expensive and/or shared resources, (ii) performs computation, and (iii) determines the next step. Accelerator calls within a single step must be independent of each other, a condition that is checked by the Gorilla compiler. Gorilla's infrastructure guarantees that each step is fully complete before the next step starts, giving the illusion of a sequential programming model to the domain expert. The results of all previous steps as well as accelerator calls are available to the current step. The same program of steps (but not necessary the same path through the program) is executed on each input data, such as a packet in a network processing application. Figure 2 shows example domain code. The accelerators themselves can be written as a domain program consisting of steps that can call other downstream accelerators. An example of such case is given in Section 4.

3.2 Canonical architecture

A canonical architecture is a highly parameterized template designed to efficiently implement a specific application class. Parameters enable the easy trading off of performance, resource sizing, and selecting options such as scheduling algorithms. The template contains code to implement any of the valid parameter values. The Gorilla tool chain combines a canonical architecture with domain code and parameter values provided by the domain expert to generate an efficient, specialized hardware implementation.

```
IPv4_check() {
  status = IPv4_header_integrity_check(Header);
  if (status == CHKSUM_OK)
    Next_step = IPv4_lookup;
  else
    Next_step = Exception;}

IPv4_lookup() {
  Da_class = lookupx.search(Header.IPv4_dstaddr);
  Sa_class = lookupy.search(Header.IPv4_srcaddr);
  if (Da_class == NOT_FOUND)
    Next_step = Exception;
  else if(Sa_class == NOT_FOUND)
    Next_step = Exception;
  else
    Next_step = IPv4_modify;}

IPv4_modify() {
  if((IP_update_fields(Header) == ZERO_TTL))
    Next_step = Exception;
  else {
    Dport = Da_class.dport;
    Next_step = Emit;
  }}
```

Figure 2: Simplified IPv4 steps

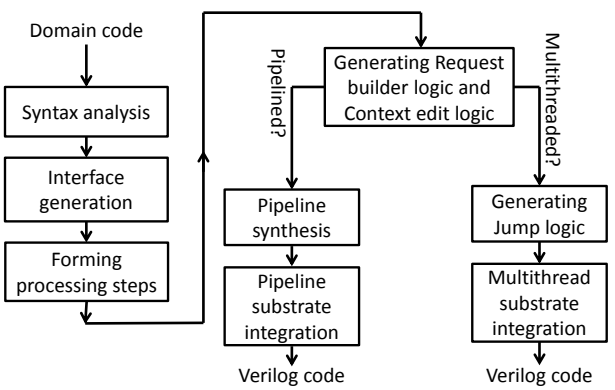

Figure 3: Compilation process for composite templates and user programmed templates

Figure 4 shows a canonical architecture discussed in Section 4.1. Though it was no simpler to write than a single implementation, it generates a wide range of possible implementations of substrates for packet processing applications.

3.3 Gorilla Tool Chain

The Gorilla tool chain consists of domain code compiler and scripts (Figure 3) and was developed using the ANTLR [1] tool and Perl. The input to the tool chain is the (i) domain code for the templates, (ii) all templates in the design and (iii) other parameter values. The compiler generates a specialized hardware implementation using these inputs.

The domain code compiler ensures that there are no dependencies or structural hazards when accessing accelerators in each step. The compiler splits a step to resolve such hazards when necessary. The compiler splits the domain code into three parts: (i) Request Builder logic, (ii) Context Edit logic, and (iii) Jump logic. Request Builder logic computes arguments for the accelerator requests and issues the requests. Context Edit logic updates thread contexts as well as global variables, using accelerator replies and state from previous steps. Jump logic determines the next step based on the results of the computations in the current step.

The code is currently compiled to Verilog that is then either inserted into a multithreaded template or is used to generate a pipeline. Multithreaded hardware supports multiple "threads", where each thread is processing a specific unit of work, such as a single packet in a network processor. Threads are switched in hardware when progress on the running thread is blocked due to long or variable latency operations or contention for resources. Multithreading enables applications to transparently trade off accelerator usage (e.g., one data element requires 10 accelerator calls to process and another requires none.) As the number of bidders (requesters) for accelerators grows, contention for shared resources increases as well, increasing variability, making multithreading more and more important. The multithreading template automatically handles all multithreading scheduling and storage issues and receives/sends the data through scratch pad memories.

As an alternative, if (i) there are only forward jumps in the domain code and (ii) accelerator delays are constant, the compiler can automatically generate a pipelined implementation from the domain code by assembling the Request Builder, and Context Edit of all processing steps into a straight pipeline. Pipelining avoids the overhead of thread scheduling and thread context memories, delivering high throughput when the maximum latency is fixed and reasonably small. Pipelining is less efficient when latencies are long, there are irregular transitions in control flow, or there are data dependencies between pipeline stages.

4. EXAMPLE: NETWORK PROCESSOR

Fine grain irregularities are common in packet processing due to heavy control flow dependency on data and contention for shared resources, making it an ideal application to demonstrate Gorilla's capabilities. We implemented a packet processing canonical architecture and a router domain code that supports IPv4, IPv6, and MPLS in Gorilla C. The IPv4, IPv6, and MPLS applications have 10, 12, and 13 steps respectively. Most of the steps fully utilize scratch pad memory (128bits wide) bandwidth. There are currently between one to six accelerator calls, depending on the packet protocol and router configuration, for each packet.

The Verilog code that was generated by the Gorilla tool chain is passed through the standard Xilinx tools for simulation and implementation on an FPGA. Two man-years were spent on building the tools and infrastructure templates. Once completed, however, the IPv6 protocol required only two man-weeks and the MPLS protocol four man-weeks.

4.1 NP canonical architecture

The canonical architecture for our Network Processor (NP) is shown in Figure 4. Every component is a highly parameterized template. When packets enter the NP, a programmable pre-processor adds necessary meta data to packet headers and splits the packets into a header and a body, passing the header through the processing pipeline and storing the body in a separate buffer. A header is assigned, in a load balanced fashion, to an arbitrary thread in an arbitrary engine. Engines which are the main packet processing elements contain hardware compiled from the domain code, creating an entirely hardware implementation of the domain

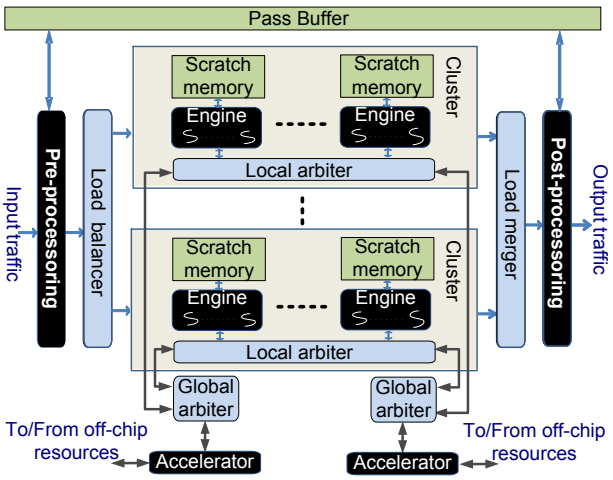

Figure 4: NP canonical architecture: Black modules are domain-specific templates. The sizing of resources as well as resource management policies are settable using parameters in the respective templates.

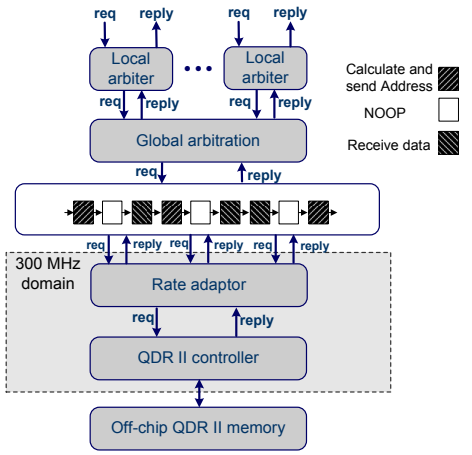

Figure 5: Simplified IPv4 trie lookup architecture including arbitration structures

code. Domain code makes calls to the hardware accelerators that are also compiled by the Gorilla compiler. Most hardware accelerators are pipelined, rather than multithreaded, since the accelerators are given dedicated resources with deterministic latencies. Once processing is complete, the header goes through a programmable post-processor to strip the meta data, get recombined with its body, and forwarded in arrival order.

Header processing domain code is compiled with a multithreaded engine template by the Gorilla compiler. It reads pre-written packet header information from scratch memories that the packet pre-processing domain code compiled with the preprocessor template has written. Special actions are taken by compiler as well as engine address translation to map the packet protocol fields into block RAM memory addresses. When the engine encounters a long latency operation, such as an accelerator call, the engine uses its multithreading support to switch to a ready thread whose accelerator calls have all returned.

Accelerators are essentially the same as in any network processor including IP lookup accelerator or flow counting accelerator. Accelerators that have dedicated off-chip QDR memory with deterministic delay are compiled to pipelined implementations. For example a trie-based lookup consists of different steps to walk through the trie levels. Each step can be written in Gorilla C to process the current node and chase the pointer to the next level node. Retrieving node information from the memory is done as an accelerator call which in this case is a memory (downstream accelerator) for our lookup accelerator.

4.2 Trie Lookup Accelerator

Many techniques have been proposed for high throughput IPv4 lookup engines [8, 12, 15, 40]. To support a large number of prefixes, Gorilla provides a lookup accelerator that stores the forwarding table in off-chip SRAM QDRII [26] that supports a new double word read operation and a new double word write operation every memory clock cycle.

We implemented a common trie lookup algorithm [12] as domain code. The algorithm divides an IPv4 address into three chunks of 20b, 4b, and 8b. At each stage, the QDR address is calculated using the current address chunk and the value returned from the QDR in the previous stage. If the entry is a leaf entry, no further read requests for that particular lookup are made. Since the pipelined lookup accelerator is clocked at 100MHz and the QDR is clocked at 300MHz, three requests from three different stages of the lookup pipeline are sent to QDR memory each 100MHz clock cycle. We use rate adaption logic between the trie lookup pipeline and QDR for this purpose. A simplified version of the generated pipelined architecture for our IPv4 trie lookup algorithm is shown in Figure 5.

Figure 6 shows the capacity requirements, in 18 bit words per entry, for different RIPE [30] routing tables. We use a single QDR chip for each QDR channel and assume a four million word QDR part.

The IPv6 lookup accelerator is similar to the IPv4 lookup accelerator except the number of trie levels is 6 instead of 3. Therefore, each IPv6 lookup accelerator has twice delay as IPv4 lookup and requires two QDR memory channels to provide full bandwidth.

The NP prototype supports multiple MPLS labels per packet. The prototype's MPLS lookup is a two level architecture with the first level acting as a cache for the second level. The first level of MPLS lookup is done using a multi-bin indexed table indexed with a hash function. If the lookup hits at the first level, it takes eight clock cycles. If the lookup misses, it is handled using exactly the same architecture as IPv4 lookup. Clusters direct their missed lookup requests to a global lookup unit which use off-chip QDR memory to lookup the MPLS labels. The MPLS lookup accelerator uses a direct mapped table with one million entries stored in QDR memories.

Our prototype's IPv4 and IPv6 lookup accelerators assume no locality and always access the QDR SRAMs for all memory reads, to ensure that full performance will always be available, regardless of the traffic pattern. Such immunity to performance "divots" is critical in high-end applications, such as core routers; otherwise, they would be highly suscep-

	Prefixes	Trie Config	Level1 entries	Level2 entries	Level3 entries	Total
rrc00, Ripe NCC Amsterdam	344,029	20,4,8	1,048,576	1,053,584	679,680	2,442,000
rrc01, Linx London	338,947	20,4,8	1,048,576	1,020,416	534,016	2,336,000
rrc02, Sfinx Paris	274,115	20,4,8	1,048,576	865,104	210,924	2,019,142
rrc16, Miami	344,029	20,4,8	1,048,576	1,051,792	352,512	2,276,624

Figure 6: Lookup unit storage requirement in 18bit words

MPPS	Accelerator operation	QDRII channels	QDRII+ channels	Required MPLS traffic - QDRII	Required MPLS traffic - QDRII+
100	IPv4 LU	1	1	0	0
100	IPv6 LU	2	1	0	0
100	IPv4 LU-FC	3	2	0	0
100	IPv6 LU-FC	5*	3	27%	0
200	IPv4 LU	2	1	0	0
200	IPv6 LU	4	2	0	0
200	IPv4 LU-FC	6*	3	42%	0
200	IPv6 LU-FC	10*	5*	69%	27%

Figure 7: QDR random transaction rate budgeting(LU: destination lookup, LU-FC: destination lookup, source lookup and flow counting), assuming maximum of four 300MHz QDR channels per FPGA configurations with * cannot fit in current FPGAs

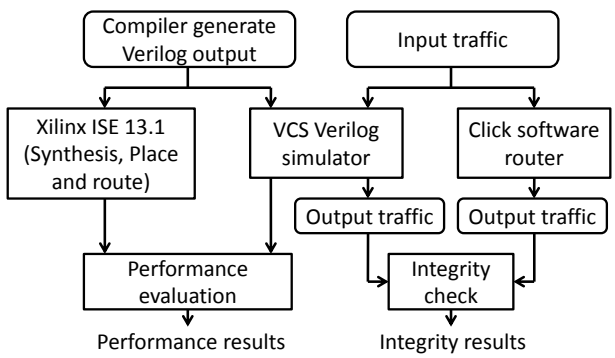

Figure 8: Simulation-Based Verification Process

tible to performance attacks. The control processor accesses QDR memories through the network processor to update IPv4, IPv6, and MPLS tables. Although our current lookup architecture can handle the required lookup throughput for our application, using compressed lookup structures and/or caching can reduce the number of off-chip transactions and, consequently, save power.

4.3 Flow Counting

Network processors in routers keep track of millions of flows for security, management, and QoS purposes [34]. In order to demonstrate Gorilla performance and programmability for flow counting, we implemented a counter scheme in which a 216 bit partitionable counter can be manipulated for each packet. That 216 bit counter can be split into three independent 72 bits counters (36 bits QDR data width $\times$ Burst length of 2.)

A pipelined flow counting architecture, which is very similar to the pipelined lookup architecture, is used for counter updates. The only difference is that writes are also required. Therefore a counter update operation includes three reads and three writes each from different stage of the pipeline. Read operations and write operations are each performed by a dedicated data channel provided by QDR SRAMs. The prototype counts the number of packets for each flow ID, which is generated by concatenating the class IDs associated with source and destination lookups. The flow ID is used as the index to the counter array. Because we support 3 72b of counter per packet, up to two other 72b counters for each packet are supported by the prototype.

4.4 FPGA Pins

Figure 7 shows the required number of QDR channels for each of the Gorilla configurations we explored. Although there are enough pins to go beyond four channels, we did not achieve post place and route timing closure for more than four channels running at 300MHz. Thus, we assume that a maximum of four 300MHz QDR channels are feasible on a single FPGA, making the entries in the table that require more than four channels infeasible using current memory technology and FPGAs. Therefore, 200MPPS packet processing with single level flow counting is memory bound. Hierarchical flow counting can address the problem, but has not yet been implemented in our system.

The QDR consortium recently announced the availability of QDRII+ memories with Random Transaction Rate(RTR) of 600M operations/sec [27] in the near future. This technology could double off-chip bandwidth and, therefore, further improve performance.

Higher throughput can be achieved providing a portion of the traffic has lower memory throughput demands. For example, an MPLS packet with only cluster level lookup does not use QDR channels, allowing other packets to use the extra bandwidth. Figure 7 shows the minimum portion of the MPLS traffic that does not require off-chip memory accesses for delivering the desired performance with both QDRII and QDRII+ standards.

4.5 NP Evaluation

We targeted three different FPGAs designed for networking applications and synthesized various configurations of Gorilla on those three FPGAs using the Xilinx ISE 13.1 tools. The Xilinx Virtex-5 TX240T FPGA (used on the NetFPGA-10G board [22]) is used to implement the IPv4-only NP. We targeted the Xilinx Virtex-6 HX380T for our 50Gbps multi-protocol (IPv4, IPv6, and MPLS) NP and the Xilinx Virtex-7 VHX870T for our 100Gbps multi-protocol NP. In all cases, the core (engines, clusters, load balancing, merging, pre-processing and post-processing) run at 100MHz, while external QDR SRAMs run at 300MHz.

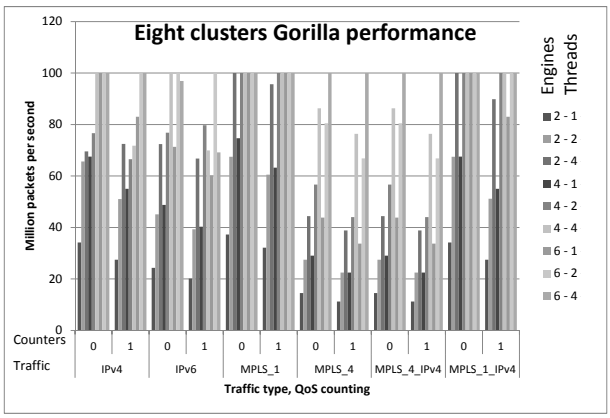

Figure 9: Performance results for 50Gbps sourced traffic with post place and route timing closure

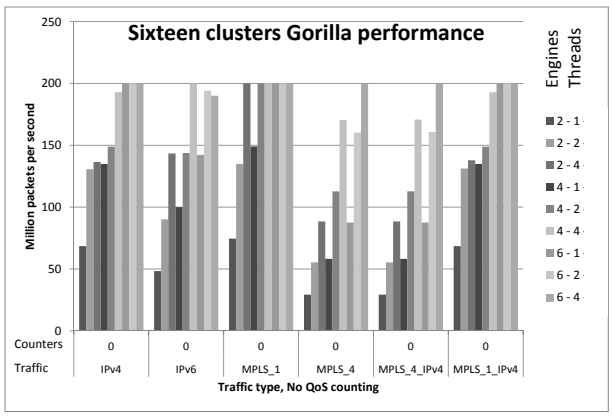

Figure 10: Performance results for 100Gbps sourced traffic with post place and route timing closure

Figure 8 shows the simulation component of our verification process for our NP family. The Click software router [17] was used as the reference system to verify the correctness of NP. In all reported performance results, the functionality of NP is checked against Click software router and post place and route timing closure is met. Synopsys VCS was used to simulate the NP for performance and functionality testing and debugging. Performance results using 100MPPS (50Gbps) and 200MPPS (100Gbps for 64 bytes packets) input traffic rates are shown in Figure 9 and Figure 10 respectively. Packets are dropped whenever performance does not reach the input traffic rate. We use clusters-engines-threads notation to represent a specific configuration. For example in a 16-8-4 configuration there are 16 clusters, 8 engines, and 4 threads.

The largest configuration which fit in Virtex-6 HX380T achieves 50Gbps (100MPPS) supporting all three protocols simultaneously and the largest configuration which fit in Virtex-7 VHX870T achieves 200MPPS supporting all three protocols simultaneously without flow counting.

IPv4 Routing: We generated the IPv4 traffic by extracting only minimum size packets from the CAIDA [3] traffic dump files. In addition to minimum size packet work loads, we tested our IPv4 router using several of equinex-chicago anonymized traffic traces from CAIDA. The major steps in the processing of IPv4 is processing layer two protocol (e.g. Ethernet), extracting and validating the IPv4 header, Looking up source and destination addresses, and modify the packet. Also for IPv4 and IPv6 packets, flow counting adds an extra Gorilla step. There are other steps to handle control packets as well as packets with integrity problem in their headers.

Figure 10 shows that 200MPPS IPv4 routing performance without flow counters can be achieved using three different 16-6-* configurations (16-4-4 configuration has some packet loss.) With flow counting, two of the configurations (16-6-2, and 16-6-4) deliver 200MPPS.

IPv6 Routing Because we did not have IPv6 traces, we IPv4 packets in our minimum-sized IPv4 traffic traces to IPv6 packets. We used a six-level trie for the lookup accelerator with random entries in the route table. Although the IPv6 program is quite similar to the IPv4 program, it reads 128bit addresses instead of 32bit addresses, putting more pressure on scratch memory. Therefore the header fields may need to be read in multiple steps. Also the trie lookup operation for IPv6 takes twice as long as IPv4 lookup.

MPLS Switching We wrote three Gorilla processing steps to extract the tag, tag lookup, and tag manipulation. These steps read the MPLS header, lookup the MPLS tag, and do the label swap operation respectively. If MPLS headers are stacked, the program jump back from the tag manipulation step to read the next tag.

We generated a variety of MPLS packets with different numbers of stacked labels ranging from one to four. As is shown in Figure 10, most of the Gorilla configurations can handle MPLS_1 traffic, that only contains MPLS packets with one label, without packet loss. MPLS_4, that contains MPLS packets with four labels, needs four tag lookups and four iterations of MPLS steps. As a result only the 16-6-4 configuration can deliver the desired performance when flow counting is off. When flow counting is turned on, the prototype delivers 195.7 MPPS for MPLS4 traffic.

Mixed Traffic In addition to homogeneous IPv4, IPv6, and MPLS test traffic, we generated mixed traffic combining different packet types to study the effect of protocol versatility on the Gorilla generated NP. For example, MPLS-1-IPv4 contains one labeled MPLS packets interleaved with IPv4 packets. When IP traffic is mixed with MPLS traffic, the performance degrades (Figure 10) due to the static inter-cluster packet scheduling algorithm we used in current prototype. We expect that a slightly more dynamic scheduler will fix this problem.

4.5.1 FPGA Resource Utilization

Figure 11 shows FPGA utilization as well as packet processing throughput using a Virtex-5 TX240T with different numbers of engines and threads. We report FPGA resource utilization for two different systems, an *Embedded test system* as well as a *NetFPGA integrated system*.

- *Embedded test system* is an NP without a framer and external memory controllers. Internal packet generators, internal statistical collectors, and internal lookup memories are used instead.
- *NetFPGA integrated system* is an NP targeting the NetFPGA-10G that includes 4*10Gbps Ethernet MAC

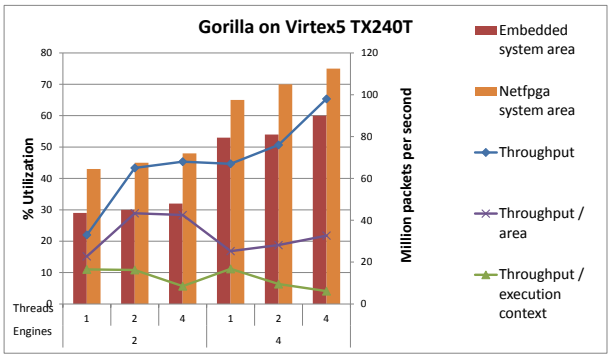

Figure 11: FPGA resource utilization and performance of Gorilla on Virtex-5 TX240T

Engines	Threads	Virtex-6 LUT utilization%	Virtex-7 LUT utilization%
Four	One	27	28
Four	Two	35	34
Four	Four	50	48
Six	One	42	39
Six	Two	53	51
Six	Four	75	73

Figure 12: FPGA utilization of Gorilla on Virtex-6 VHX380T, Clusters=8 and Virtex-7 855T, Clusters=16

controllers connected to FPGA Multi-Gigabit I/Os. This system includes necessary QDR-II controllers for lookup and flow counter accelerators. The integration of the NP with all necessary NetFPGA-10G IP demonstrates the fact that a single FPGA IPv4 router including MAC controllers is feasible.

Figure 11 shows that an 8-4-4 configuration (which delivers 100MPPS) fits in a NetFPGA integrated system with Virtex-5 TX240T. In addition to raw throughput, the figure shows throughput per execution context as well as normalized throughput per area. As is expected, for a particular number of engines, increasing number of threads improves the throughput per area while increasing the number of engines reduces the throughput per area.

Figure 12 shows the FPGA resource utilization of the embedded test system on both Virtex-6 VHX380T, and Virtex-7 855T. We explored an extensive amount of the design space to meet the post place and route timing closures for these two FPGAs. This exploration was only possible due to the parametric nature of the templates in the system.

4.5.2 *Scheduling, Arbitration, and Reordering*

The Gorilla resource management templates enables easy trading off between area and performance. Figure 13 shows the performance of Gorilla running IPv4 packet routing using two different lookup delays (32 and 128 cycles), and two different thread schedulers. When the number of threads is increased, performance saturates earlier with low latency accelerators than with high latency accelerators. A *round robin* thread scheduler switches to the next thread whenever a thread accesses an accelerator, regardless of whether the next thread is ready or not. A *ready-to-execute* thread

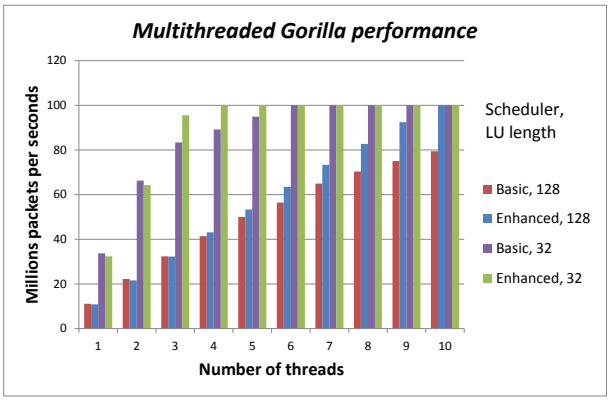

Figure 13: Multithreaded Gorilla throughput scaling for IPv4 with 100MPPS source rate (clusters=8, engines=2)

scheduler switches to a thread which is not stalled every cycle. Although the ready-to-execute thread scheduler performs better for large number of threads, it requires a reorder queue because processing of packets might finish out of order. Consequently, ready-to-execute thread scheduling imposes a considerable area overhead on the design (10%-15% for large configurations.) When accelerator latencies are small, round robin scheduling saves area. However for large latency accelerators we need ready-to-execute scheduling for performance scaling.

Another parametric template in the Gorilla infrastructure is the arbitration structure. Arbitration for accesses to global accelerators is done in a hierarchical fashion. Either fair, yet complex arbiters or low overhead unfair arbiters are possible. Due to the large contention on global arbiters comparing to local arbiters, in most cases strong fairness is only required for global arbiters. This works perfectly for our design because of two reasons. Firstly, the local schedulers, like the thread scheduler and the engine scheduler, are self throttling, ensuring freedom from starvation even with an unfair scheduler. Secondly, there are many more local arbiters than global arbiters. Consequently there are many small, lightweight arbiters and few large arbiters in the design. Using fair local arbiters in a large configuration imposes an 8% area overhead on the whole design.

4.6 Board Implementation

Since we did not have access to high performance networking test gear, our real implementation included an embedded test platform on the FPGA to generate random packets and verify that they were processed correctly. Note that we met timing closure with exactly the same network processor integrated with real Gigabit transceivers and memory controllers on equivalent FPGA with real I/Os, indicating we will run correctly with real interfaces and memory.

We implemented a 16-3-2 configuration of IPv4-only Gorilla with embedded peripherals on a Xilinx Virtex 6 ML605 board [37]. The implementation includes an embedded packet generator unit, an embedded packet collector, and a statistic report generator. The lookup engines (Section 4.2) are using a BRAM-based module which exactly emulates the QDR timing. The working board validates the functionality

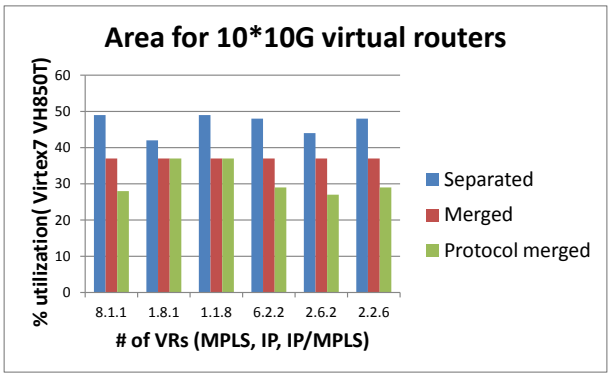

Figure 14: Protocol-Based Virtual Router Consolidation in Gorilla

of NP on real hardware and that the delivered performance is consistent with the Verilog simulation results. The board handles 40Gbps (100MPPS) traffic (injecting a new packet every clock cycle) using 55% of LUT resources.

Using the Xilinx *system monitor* tool, the measured current and voltage driving the FPGA core logic (VCCint and the corresponding driving current) indicated power is less than 4 watts. Although it does not include I/O, the power is stunningly low. The core logic power of Gorilla is comparable or better than the power consumption reported for state of the art network processors [23,28]. The I/Os should be similar, implying that our total solution will be at least power competitive.

4.7 Performance Analysis

The network processor achieves its high performance from a combination of reasons. Since the accelerators are the most critical resources in the system, as long as the accelerators always have work to do, the system's throughput is maximized. Hardware-implemented finite state machines issue calls very quickly, ensuring that each thread is producing as many requests to accelerators as possible. Aggressive multithreading with hardware-implemented synchronization of accelerator calls maximizes the number of ready threads which, in turn, maximizes the number of accelerator calls **and** makes writing the domain code easy and sequential. Gorilla engines can issue a full set of accelerator calls **every** cycle, compared to tens of cycles needed for a single accelerator call on a more conventional, instruction-based network processor. Accelerators are themselves designed to maximize throughput. The end result is a system that maximizes the number of accelerator calls and, therefore, performance.

The aggressive parameterization of the templates enables rapid, extensive exploration of the implementation space, enabling the search for an optimized configuration in a reasonable time and accommodating last minute changes. Thus, templates can dramatically improve productivity.

5. VIRTUAL ROUTER CONSOLIDATION

Router virtualization is a technique to share the same packet processing hardware across different virtual routers to reduce capital and operational expenditures. FPGAs provide unique opportunity for building hardware virtual routers. Different techniques have been proposed for consolidating virtual routers on a single FPGA [19, 39]. A router virtualization framework should provide three important characteristics (i) isolated performance, (ii) reconfigurability, and (iii) scalability in terms of number of virtual routers.

We do not present a full fledged virtualization framework. Instead, we demonstrate how a flexible and modular network processor design using Gorilla can consolidate virtual routers efficiently. Consolidating virtual routers can be done either using isolated hardware or merged hardware. For example, when 10 10Gbps virtual routers are consolidated, 10 isolated routers can be instantiated, each handling its own traffic flow, or a single 100Gbps router that handles the merged flows. In a hardware router in which resources are allocated for worse case traffic patterns, individual routers can be merged while maintaining performance isolation for each virtual router.

Given a set of virtual routers, each supporting different protocols, it is possible to create a customized network processor that supports all the virtual routers with guaranteed throughput and minimal resource utilization. When virtual routers with the same protocol(s) are merged, area efficient techniques like multithreading and pipelining help to improve performance per area comparing to an isolated configuration. When virtual routers implementing different protocols are merged, one can support all the protocols by simply instantiating engines that support all required protocols. Although this method might save area by reusing common structures, resources are used inefficiently because the engines are more general than they need to be.

Our solution, *protocol-based consolidation*, uses a heuristic algorithm where groups of virtual routers are merged only if the virtual routers have some common protocol. Merging is only considered if the merged area is less than the area of isolated implementation of virtual routers. We demonstrate the result of such protocol-based consolidation by implementing 10*10G virtual routers using Gorilla. Each of the virtual routers supports one of the following: MPLS only, IPv4 only, or both MPLS and IPv4 protocols. Only single label MPLS processing is considered in this study. Six different virtual router configurations are considered. For example, in the 8.1.1 configuration there are eight MPLS-only virtual routers, one IPv4-only virtual router, and one IPv4/MPLS virtual router.

Figure 14 shows the LUT utilization of each of these configurations on Virtex-7 855T using isolated, merged, and protocol-based consolidation methods. When the protocol-based consolidation method is used on an eight MPLS, one IPv4, and one IPv4/MPLS virtual router configuration, the MPLS only routers are merged but not the other two virtual routers. However for the one MPLS, eight IPv4, and one IP/MPLS configuration all virtual routers are merged. This is due to the fact that for 80Gbps a single label MPLS processing requires fewer engines and fewer threads per engine than IPv4 processing. As the result, isolating a 80Gbps MPLS router from the other two routers saves a considerable area. On the other hand, when building a 80Gbps IPv4 router, the incremental area overhead for equipping engines with MPLS processing steps is smaller than adding two isolated 10G MPLS routers. Overall, for the sample virtual router configurations, protocol-based consolidation saves on average 33% of area comparing to the isolated method and 15% of area comparing to the normal merged method.

6. RELATED WORK

The architectural techniques including using accelerators, course grain pipelining, multithreading, parallelism among multiple engines, and hierarchical arbitration have been used extensively in the past. However, to the best of our knowledge, they have not been combined the Gorilla way. Also, they have not been automatically generated from stylized C or utilized FPGAs to provide programmability. The exception is our own patented work, that was not evaluated in the patent [6].

Templates (called different names in different languages including "templates" in C++ and "macros" in Lisp) have long been used in software to improve reuse. Effective parameterized hardware, though explored repeatedly in the past, has not yet achieved widespread acceptance. Recent research [24, 33] has demonstrated the power of templates, especially for high performance and/or low power designs. Gorilla differs in that it separates the functionality (application) from the parameterized template while others are more of a parameterized implementation. Others [25] have separated functionality and the micro-architecture, but do not focus on high parameterization of the micro-architecture (template.)

Requiring the user to provide a template in addition to domain code that describes functionality is Gorilla's key differentiator from traditional C-to-gates tools, such as AutoESL [2] and CatapultC [4], that attempt to infer the appropriate hardware structure. Some researchers tried to incorporate the architectural information while synthesizing the high level code but their method is limited to generating single processing engine on a pre-defined datapath [29].

Among many publications related to automatic compilation of applications into hardware, Convey [36] and Optimus [14] are perhaps the most similar to Gorilla. Convey provides predefined building blocks for a particular application personality to domain code. The tool chain compiles domain code to pipelined implementations called systolic structures. Temporal common sub-expression elimination, or loops in the pipeline, is used to reduce area overhead. Gorilla is more free formed than Convey, implementing state machines rather than application-specific processors, and focuses on balancing the performance-area trade off by sizing high level resources like engines and clusters. Optimus maps StreamIt [35] applications, that are described using streaming graph and filter kernels, to FPGAs. Gorilla targets applications with more control flow irregularity and shared resource access irregularity.

Many projects aim to improve the programmability problem for high performance FPGA-based packet processing applications. Some focus on the programmability aspect, without achieving high performance [18, 31]. NetFPGA [20] uses a set of libraries and reference designs in order to simplify the implementation of new packet processing systems. However a hardware expert is needed to design the entire hardware implementation. A Xilinx project targets 100Gbps network processing and traffic management in a pair of FPGAs, but is designed in a traditional way and is not yet available [11]. Another project [16] demonstrated 40Gbps in a FPGA for MPLS-only packet processing without any high level programmability features. Gorilla generated network processors, on the other hand, can handle 200MPPS (64B packets) for any mix of IPv4, IPv6, and MPLS on a single Virtex7 FPGA and it is programmable with C.

Routebricks [7] is a parallelized version of the Click software router and is the fastest pure software router we are aware of. Routebricks runs at 23.4M packets per second (12Gbps with 64B packets) on four systems, each containing eight Nehalem cores for a total of 32 cores. Routebricks supports four flows of 3Gbps each, a significantly easier problem than the single 100Gbps flow Gorilla supports. Gorilla also has deterministic performance and resiliency against adversarial traffic, and consumes much less power.

Packetshader [32] achieves 40Gbps on a system with two quad-core Nehalem processors, 12GB of memory, two I/O hubs and two NVIDIA GTX480 cards. Packetshader performance is dependent on intelligent NICs that balance load between cores and is currently limited by PCIe performance. The load balancers make PacketShader venerable to adversarial traffic. The partitioning introduces overhead to keep common state between cores. In addition, each GTX480 consumes up to 250W.

There are many custom network processors including those from Cavium [5], EZChip [9], Xelerated [38], Freescale [10], and LSI [21]. Based on their processor data sheets, Gorilla's performance is roughly the same as EZChip's and Xelerated's recently announced 100Gbps network processors and faster than all other vendors. A Gorilla NP is more flexible than those solutions because it is implemented entirely on an FPGA and is also at least as easy to program. From architectural perspective the difference between Gorilla and other network processors is that the hardware is specialized for the application and parallelization is expressed and managed using structures which are scalable in FPGAs.

7. CONCLUSIONS AND FUTURE WORK

We describe the Gorilla programming model, templates, canonical architectures, and tool chain. Gorilla separates the work of domain experts from the work of hardware experts, enabling each to work in their area of expertise and automatically and efficiently combining that work.

We use Gorilla to generate a family of network processors capable of handling almost all combinations of MPLS, IPv4, and IPv6 traffic at 200MPPS rate in a single FPGA. The packet processing engines, accelerators, packet splitter and reassembly are written by domain experts in a subset of sequential C, automatically compiled to hardware, and then merged with parameterized templates written by hardware experts. Domain experts only need to concentrate on their domain and can safely ignore hardware and parallelization issues. Hardware experts, on the other hand, only need to focus on hardware, rather than having to know the details of the implementation.

We studied the required FPGA resources using three modern FPGAs to demonstrate that FPGAs have enough logic, memory, and IO resources to deliver the required packet processing performance for the mentioned applications. We reported the performance results for various traffic loads and processor configurations.

Although we focused on packet processing as the proof of concept for Gorilla methodology, we believe that the same methodology can be applied on other data parallel applications like genome sequencing, or hardware simulation acceleration. We plan to port more application spaces and applications to Gorilla and to build a router using the Gorilla generated network processor described in this paper.

8. ACKNOWLEDGEMENTS

This material is based upon work supported in part by the National Science Foundation under Grants 0747438 and 0917158. We acknowledge the anonymous reviewers for their insightful comments.

9. REFERENCES

[1] Antlr compiler compiler tool. http://www.antlr.org/.

[2] AutoESL. http://www.autoesl.com/.

[3] The cooperative association for internet data analysis. www.caida.org.

[4] CatapultC. http://www.mentor.com/.

[5] Cavium network processor. http://datasheet.digchip.com/227/227-04668-0-IXP2800.pdf.

[6] Dennison, L., and Chiou, D. Compilable, reconfigurable network processora simulation method. United States Patent 7,823,091, Oct. 2010.

[7] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. Routebricks: exploiting parallelism to scale software routers. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 15–28.

[8] Eatherton, W., Varghese, G., and Dittia, Z. Tree bitmap: hardware/software ip lookups with incremental updates. *SIGCOMM Comput. Commun. Rev. 34* (April 2004), 97–122.

[9] Ezchip np-4 network processor. http://www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf.

[10] Freescale c-5 network processor. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=C-5.

[11] Reconfigurable computing for high performance networking applications. http://www.xilinx.com/innovation/research-labs/keynotes/Arc2011-Keynote.pdf.

[12] Gupta, P., Lin, S., and Mckeown, N. Routing Lookups in Hardware at Memory Access Speeds. In *In Proceedings of INFOCOM* (1998), pp. 1240–1247.

[13] Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., and Horowitz, M. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 37–47.

[14] Hormati, A., Kudlur, M., Mahlke, S., Bacon, D., and Rabbah, R. Optimus: efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2008), pp. 41–50.

[15] Jiang, W., and et al. Parallel IP Lookup using Multiple SRAM-based Pipelines, 2008.

[16] Karras, K., Wild, T., and Herkersdorf, A. A folded pipeline network processor architecture for 100 gbit/s networks. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (New York, NY, USA, 2010), pp. 2:1–2:11.

[17] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[18] Kulkarni, C., Brebner, G., and Schelle, G. Mapping a domain specific language to a platform fpga. In *Proceedings of the 41st annual Design Automation Conference* (New York, NY, USA, 2004), DAC '04, ACM, pp. 924–927.

[19] Le, H., Ganegedara, T., and Prasanna, V. K. Memory-efficient and scalable virtual routers using fpga. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (2011), pp. 257–266.

[20] Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education* (June 2007).

[21] Lsi advanced payload plus network processor. http://www.lsi.com/networking_home/networking-products/network-processors/.

[22] NetFPGA 4*10G board . http://netfpga.org/foswiki/NetFPGA/TenGig/Netfpga10gInitInfoSite.

[23] Netronome flow processor. http://www.netronome.com/pages/network-flow-processors.

[24] Ng, M. C., Vijayaraghavan, M., Dave, N., Arvind, Raghavan, G., and Hicks, J. From WiFi to WiMAX: Techniques for High-Level IP Reuse across Different OFDM Protocols. In *Proceedings of MEMOCODE 2007* (2007).

[25] Patil, N. A., Bansal, A., and Chiou, D. Enforcing architectural contracts in high-level synthesis. In *DAC* (2011), pp. 824–829.

[26] 300Mhz Two Word Burst QDRII datasheet. http://www.cypress.com/?docID=21484.

[27] QDR plus pre-announcement by QDR consortium. http://eetimes.com/electronics-products/electronic-product-reviews/memory-products/4215361/QDR-Consortium-pre-announces-new-SRAMs.

[28] Cisco QuantumFlow architecture. http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.pdf.

[29] Reshadi, M., Gorjiara, B., and Gajski, D. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *Proceedings of the 2005 International Conference on Computer Design* (2005), pp. 69–76.

[30] RIPE Route Information Service. http://www.ripe.net/projects/ris/rawdata.html.

[31] Rubow, E., McGeer, R., Mogul, J., and Vahdat, A. Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware. In *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on* (Oct. 2010), pp. 1 –10.

[32] Sangjin Han, Keon Jang, K. P., and Moon, S. PacketShader: a GPU-accelerated Software Router. In *in Proc. of ACM SIGCOMM 2010, Delhi, India* (2010).

[33] Shacham, O., Azizi, O., Wachs, M., Qadeer, W., Asgar, Z., Kelley, K., Stevenson, J., Richardson, S., Horowitz, M., Lee, B.,

Solomatnikov, A., and Firoozshahian, A. Rethinking digital design: Why design must change. *Micro, IEEE 30*, 6 (nov.-dec. 2010), 9 –24.

[34] Shah, D., Iyer, S., Prabhakar, B., and McKeown, N. Maintaining statistics counters in router line cards. *IEEE Micro 22* (January 2002), 76–81.

[35] Thies, W., Karczmarek, M., and Amarasinghe, S. P. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction* (London, UK, 2002), Springer-Verlag, pp. 179–196.

[36] Villarreal, J., and et al. Programming the Convey HC-1 with ROCCC 2.0, 2010.

[37] Xilinx ML605 reference design board. `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm`.

[38] Xelerated. `http://www.xelerated.com/`.

[39] Yin, D., Unnikrishnan, D., Liao, Y., Gao, L., and Tessier, R. Customizing virtual networks with partial fpga reconfiguration. *SIGCOMM Comput. Commun. Rev. 41* (2011), 125–132.

[40] Zheng, K., Hu, C., Lu, H., and Liu, B. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Trans. Netw. 14* (August 2006), 863–875.