ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware

Bojie Li[§†] Kun Tan[†] Layong (Larry) Luo[‡] Yanqing Peng[•†] Renqian Luo[§†]
Ningyi Xu[†] Yongqiang Xiong[†] Peng Cheng[†] Enhong Chen[§]
[†]Microsoft Research [§]USTC [‡]Microsoft [•]SJTU

ABSTRACT

Highly flexible software network functions (NFs) are crucial components to enable multi-tenancy in the clouds. However, software packet processing on a commodity server has limited capacity and induces high latency. While software NFs could scale out using more servers, doing so adds significant cost. This paper focuses on accelerating NFs with programmable hardware, *i.e.*, FPGA, which is now a mature technology and inexpensive for datacenters. However, FPGA is predominately programmed using low-level hardware description languages (HDLs), which are hard to code and difficult to debug. More importantly, HDLs are almost inaccessible for most software programmers. This paper presents ClickNP, a FPGA-accelerated platform for highly flexible and high-performance NFs with commodity servers. ClickNP is highly flexible as it is completely programmable using high-level C-like languages, and exposes a modular programming abstraction that resembles Click Modular Router. ClickNP is also high performance. Our prototype NFs show that they can process traffic at up to 200 million packets per second with ultra-low latency ($< 2\mu$s). Compared to existing software counterparts, with FPGA, ClickNP improves throughput by 10x, while reducing latency by 10x. To the best of our knowledge, ClickNP is the first FPGA-accelerated platform for NFs, written completely in high-level language and achieving 40 Gbps line rate at any packet size.

CCS Concepts

•**Networks** → **Middle boxes / network appliances;** *Data center networks;* •**Hardware** → *Hardware-software codesign;*

Keywords

Network Function Virtualization; Compiler; Reconfigurable Hardware; FPGA

SIGCOMM '16, August 22–26, 2016, Florianopolis, Brazil
 ISBN 978-1-4503-4193-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934897

1. INTRODUCTION

Modern multi-tenant datacenters provide shared infrastructure for hosting many different types of services from different customers (*i.e.*, tenants) at a low cost. To ensure security and performance isolation, each tenant is deployed in a *virtualized network* environment. Flexible network functions (NFs) are required for datacenter operators to enforce isolation while simultaneously guaranteeing Service Level Agreements (SLAs).

Conventional hardware-based network appliances are not flexible, and almost all existing cloud providers, *e.g.*, Microsoft, Amazon and VMWare, have been deploying software-based NFs on servers to maximize the flexibility [23, 30]. However, software NFs have two fundamental limitations – both stem from the nature of software packet processing. First, processing packets in software has limited capacity. Existing software NFs usually require multiple cores to achieve 10 Gbps rate [33, 43]. But the latest network links have scaled up to 40∼100 Gbps [11]. Although one could add more cores in a server, doing so adds significant cost, not only in terms of capital expense, but also more operational expense as they are burning significantly more energy. Second, processing packets in software incurs large, and highly variable latency. This latency may range from tens of microsecond to milliseconds [22,33,39]. For many low latency applications (*e.g.*, stock trading), this inflated latency is unacceptable.

To overcome the limitations of software packet processing while retaining flexibility, recent work has proposed accelerating NFs using Graphics Processing Units (GPUs) [26], network processors (NPs) [2, 5], or reconfigurable hardware (*i.e.*, Field Programmable Gate Arrays, or FPGAs) [24, 36, 42]. Compared to GPU, FPGA is more power-efficient [19, 28]. Compared to specialized NPs, FPGA is more *versatile* as it can be virtually reconfigured with any hardware logic for any service. Finally, FPGAs are inexpensive and being deployed at scale in datacenters [24, 40].

In this work, we explore the opportunity to use FPGA to accelerate software NFs in datacenters. The main challenge to use FPGA as an accelerator is *programmability*. Conventionally, FPGAs are programmed with hardware description languages (HDLs), such as Verilog and VHDL, which expose only low level building blocks like gates, registers, multiplexers and clocks. While the programmer can manually

tune the logic to achieve maximum performance, the programming complexity is huge, resulting in low productivity and debugging difficulties. Indeed, the lack of programmability of FPGA has kept the large community of software programmers away from this technology for years [15].

This paper presents ClickNP, an FPGA-accelerated platform for highly flexible and high-performance NF processing on commodity servers. ClickNP addresses the programming challenges of FPGA in three steps. First, ClickNP provides a modular architecture, resembling the well-known Click model [29], where a complex network function is composed using simple, well-defined elements [1]. Second, ClickNP elements are written with a high-level C-like language and are cross-platform. ClickNP elements can be compiled into binaries on CPU or low-level hardware description language (HDL) for FPGAs, by leveraging commercial high-level synthesis (HLS) tools [1,6,9]. Finally, we develop a high-performance PCIE I/O channel that provides high-throughput and low latency communications between elements running on CPU and FPGA. This PCIE I/O channel not only enables joint CPU-FPGA processing – allowing programmers to partition their processing freely, but also is of great help for debugging, as a programmer may easily run an element in question on the host and use familiar software tools to diagnose.

ClickNP employs a set of optimization techniques to effectively utilize the massive parallelisms in FPGA. First of all, ClickNP organizes each element into a logic block in FPGA and connects them with first-in-first-out (FIFO) buffers. Therefore, all these element blocks can run in full parallel. For each element, the processing function is carefully written to minimize the dependency among operations, which allows the HLS tools to generate maximum parallel logics. Further, we develop *delayed write* and *memory scattering* techniques to address the read-write dependency and pseudo-memory dependency, which cannot be resolved by existing HLS tools. Finally, we carefully balance the operations in different stages and match their processing speed, so that the overall throughput of pipelines is maximized. With all these optimizations, ClickNP achieves high packet processing throughput up to 200 million packets per second [2], with ultra-low latency ($< 2\mu$s for any packet size in most applications). This is about a 10x and 2.5x throughput gain, compared to state-of-the-art software NFs on CPU and CPU with GPU acceleration [26], while reducing the latency by 10x and 100x, respectively.

We have implemented the ClickNP tool-chain, which can integrate with various commercial HLS tools [1,9]. We have implemented about 100 common elements, 20% of which are re-factored straightforwardly from Click. We use these elements to build five demonstration NFs: (1) a high-speed traffic capture and generator, (2) a firewall supporting both exact and wildcard matching, (3) an IPSec gateway, (4) a Layer-4 load balancer that can handle 32 million concurrent flows, and (5) a pFabric scheduler [12] that performs strict priority flow scheduling with 4-giga priority classes. We evaluate these network functions on a testbed with Dell servers and Altera Stratix V FPGA boards [40]. Our results show that all of these NFs can be greatly accelerated by FPGA and saturate the line rate of 40Gbps at any packet size with very low latency and neglectable CPU overhead.

[1]This is also where our system name, *Click Network Processor*, comes from.

[2]The actual throughput of a ClickNP NF may be bound by the Ethernet port data rate.

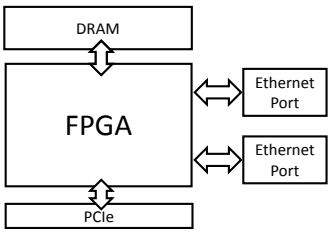


Figure 1: A logic diagram of a FPGA board.

In summary, the contributions of this paper are: (1) the design and implementation of ClickNP language and tool-chain; (2) the design and implementation of high-performance packet processing modules that are running efficiently on FPGA; (3) the design and evaluation of five FPGA-accelerated NFs. To the best of our knowledge, ClickNP is the first FPGA-accelerated packet processing platform for general network functions, written completely in high-level language and achieving a 40 Gbps line rate.

2. BACKGROUND

2.1 FPGA architecture

As the name indicates, FPGA is a sea of *gates*. The basic building block of FPGA is *logic element (LE)*, which contains a Look-up Table (LUT) and a few registers. The LUT can be programmed to compute any combinational logic and registers are used to store states. Besides basic LEs, FPGA also contains Block RAMs (BRAMs) to store data, and Digital Signal Processing (DSP) components for complex arithmetic operations. Normally, FPGAs are attached to a PC through a PCIe add-in board, which may also contain a DRAM of multi-giga bytes and other communication interfaces, *e.g.*, 10G/40G Ethernet ports. Figure 1 shows a logic diagram of a FPGA board.

Compared to CPU or GPU, FPGAs usually have a much lower clock frequency and a smaller memory bandwidth. For example, typical clock frequency of a FPGA is about 200MHz, more than an order of magnitude slower than CPU (at 2$\sim$3 GHz). Similarly, the bandwidth to a single Block memory or external DRAM of FPGA is usually 2$\sim$10 GBps, while the memory bandwidth is about 40 GBps of Intel XEON CPU and 100 GBps for a GPU. However, the CPU or GPU have only limited cores, which limits parallelism. FPGAs have a massive amount of parallelism built-in. Modern FPGAs may have millions of LEs, hundreds K-bit registers, tens of M-bits of BRAM, and thousands of DSP blocks. In theory, each of them can work in parallel. Therefore, there could be thousands of parallel "*cores*" running simultaneously inside a FPGA chip. Although the bandwidth of a single BRAM may be limited, if we access the thousands of BRAMs in parallel, the aggregate memory bandwidth can be

multiple TBps! Therefore, to achieve high performance, a programmer must fully utilize this massive parallelism.

Conventionally, FPGAs are programmed using HDLs like Verilog and VHDL. These languages are too low level, hard to learn and complex to program. As a consequence, the large community of software programmers has stayed away from FPGA for years [15]. To ease this, many high level synthesis (HLS) tools/systems have been developed in both industry and academia that try to convert a program in high level language (predominately C) into HDLs. However, as we will show in the next subsection, none of them is suitable for network function processing, which is the focus of this work.

2.2 Programming FPGA for NFs

Our goal is to build a versatile, high performance network function platform with FPGA-acceleration. Such a platform should satisfy the following requirements.

Flexibility. The platform should be *fully programmed using high-level languages.* Developers program with high-level abstractions and familiar tools, and have similar programming experience as if programming on a multi-core processor. We believe this is a necessary condition for FPGA to be accessible to most software programmers.

Modularized. We should support a *modular architecture* for packet processing. Previous experiences on virtualized NFs have demonstrated that a right modular architecture can well capture many common functionalities in packet processing [29, 33], making them easy to reuse in various NFs.

High performance and low latency. NFs in datacenters should handle a large amount of packets flowing at the line-rates of 40/100 Gbps with ultra-low latency. Previous work has shown [44] that even a few hundred microseconds of latency added by NFs would have negative impacts on service experience.

Support joint CPU/FPGA packet processing. We'd say FPGA is no panacea. As inferred from the FPGA architecture discussed earlier in §2.1, not all tasks are suitable for FPGA. For example, algorithms that are naturally sequential and processing that has very large memory footprint with low locality, should process better in CPU. Additionally, FPGA has a strict area constraint. That means you cannot fit an arbitrarily large logic into a chip. Dynamically swapping FPGA configurations without data plane interruption is very difficult, as the reconfiguration time may take seconds to minutes, depending on the FPGA's size. Therefore, we should support fine-grained processing separation between CPU and FPGA. This requires high-performance communication between CPU and FPGA.

No of existing high level programming tools for FPGA satisfy all aforementioned requirements. Most HLS tools, *e.g.*, Vivado HLS [9], are only auxiliary tools for HDL tool chains. Instead of directly compiling a program into FPGA images, these tools generate only hardware modules, *i.e.*, IP cores, which must be manually embedded in a HDL project and connected to other HDL modules – a mission impossible for most software programmers.

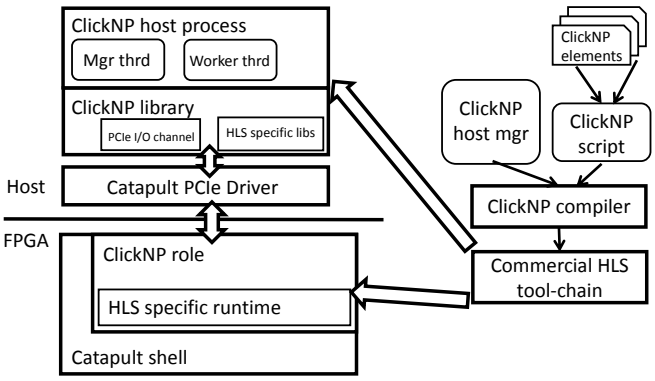


Figure 2: The architecture of ClickNP.

Altera OpenCL, however, may directly compile an OpenCL program to FPGA [1]. However, the OpenCL programming model is directly derived from GPU programming and is not modularized for packet processing. Further, OpenCL does not support joint packet processing between CPU and FPGA very well: First, communication between a host program and a kernel in FPGA must always go through the onboard DDR memory. This adds non-trivial latency and also causes the on-board memory a bottleneck. Second, OpenCL kernel functions are *called* from the host program. Before a kernel terminates, the host program cannot control the kernel behavior, *e.g.* setting new parameters, nor reading any kernel state. But NFs face a continuous stream of packets and should be always running.

Click2NetFPGA [41] provides a modular architecture by directly compiling a Click modular router [29] program into FPGA. However, the performance of [41] is much lower (two orders of magnitude) than what we report in this paper, as there are several bottlenecks in their system design (*e.g.*, memory and packet I/O) and they also miss several important optimizations to ensure fully pipelined processing (as discussed in §4). Additionally, [41] does not support FPGA/CPU joint processing and thus unable to update configuration or read states while data plane is running.

In the following, we will present ClickNP, a novel FPGA-accelerated network function platform that satisfies all aforementioned four requirements.

3. ARCHITECTURE

3.1 System architecture

Figure 2 shows the architecture of ClickNP. ClickNP builds on the Catapult Shell architecture [40]. The *shell* contains many reusable bits of logic that are common for all applications and abstracts them into a set of well-defined interfaces, *e.g.*, PCIe, Direct Memory Access (DMA), DRAM Memory Manage Unit (MMU), and Ethernet MAC. The ClickNP FPGA program is synthesized as a Catapult *role*. However, since ClickNP relies on commodity HLS tool-chains to generate FPGA HDL, and different tools may generate their own (and different) interfaces for the resources managed by the shell, we need a shim layer, called *HLS-specific runtime*, to

perform translations between HLS specific interfaces to the shell interfaces.

A ClickNP host process communicates with the ClickNP role through the ClickNP library, which further relies on the services in Catapult PCIe driver to interact with FPGA hardware. The ClickNP library implements two important functions: (1) It exposes a PCIe channel API to achieve high-speed and low latency communications between the ClickNP host process and the role; (2) It calls several HLS specific libraries to pass initial parameters to the modules in the role, as well as control the start/stop/reset of these modules. The ClickNP host process has one manager thread and zero or multiple worker threads. The manager thread loads the FPGA image into the hardware, starts worker threads, initializes ClickNP elements in both FPGA and CPU based on the configuration, and controls their behaviors by sending *signals* to elements at runtime. Each worker thread may process one or more modules if they are assigned to CPU.

3.2 ClickNP programming

3.2.1 Abstraction

ClickNP provides a modular architecture and the basic processing module is called an *element*. A ClickNP element has the following properties:

- Local states. Each element can define a set of local variables that are only accessible inside the element.
- Input and output ports. An element can have any number of input or output ports.
- Handler functions. An element has three handler functions: (1) an initialization handler, which is called once when the element starts, (2) a processing handler, which is continuously called to check input ports and process available data, and (3) a signal handler, which receives and processes the commands (*signals*) from the manager thread in the host program.

An output port of an element can connect to an input port of another element through a *channel*, as shown in Figure 3(a). In ClickNP, a channel is basically a FIFO buffer that is written to one end and read from the other end. The data unit of the read/write operations to a channel is called *flit*, which has a fixed size of 64 bytes. The format of a flit is shown in Figure 3(b). Each flit contains a header for meta-data and a payload of 32 bytes. A large piece of data, *e.g.*, a full-sized packet, is broken into multiple flits, when flowing among ClickNP elements. The first flit is marked with **sop** (start of packet), and the last flit is marked with **eop** (end of packet). If the size of the data piece is not 32, the **pad** field of the last flit indicates how many bytes have been padded to the payload. We note that breaking large data into flits not only reduces latency, but also potentially increases parallelism as different flits of a packet may be processed at different elements simultaneously. Finally, to fulfill a network function, multiple ClickNP elements can be interconnected to form a directed processing graph, which is called a ClickNP *configuration*.

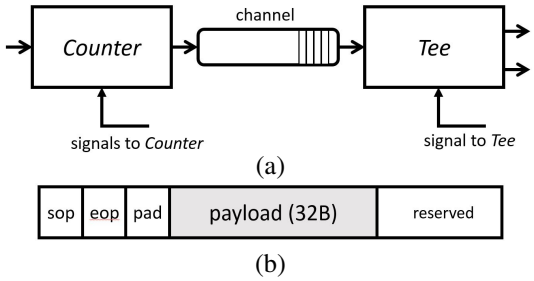


Figure 3: (a) Two ClickNP elements are connected through a channel. (b) The format of a flit.

Clearly, the ClickNP programming abstraction largely resembles Click software router [29]. However, there are three fundamental differences which make ClickNP more suitable for FPGA implementation: (1) In Click, edges between elements are C++ function calls and a *queue* element is required to store packets. However, in ClickNP, an edge actually represents a FIFO buffer that can hold actual data. Additionally, ClickNP channels break the data dependency among elements and allow them to run in parallel. (2) Unlike Click, where each input/output port can be either *push or pull*, ClickNP has unified these operations: An element can only *write (push)* to any output port, while *read (pull)* can do so from any input port. (3) While Click allows an element to directly call methods of another element (via flow-based router context), in ClickNP, the coordination among elements is *message-based*, *e.g.*, a requester sends a request message to a responder and gets a response via another message. Message-based coordination allows more parallelism and is more efficient in FPGA compared to coordination through shared memory, where accessing a shared memory location has to be serialized and would become a bottleneck.

3.2.2 Language

ClickNP elements are alike objects in an object-oriented language, and can be defined using such languages, *i.e.*, C++. Unfortunately, many existing HLS tools support only C. To leverage the commercial HLS tools, we could write a compiler that converts a object-oriented language, *e.g.* C++, to C. But this effort is non-trivial. In this work, we take an alternative path to extend C language to support element declaration. Figure 4(a) shows a code snippet of element *Counter*, which simply counts how many packets have passed. An element is defined by **.element** keyword, followed by the element name and the number of input/output ports. The keyword **.state** defines the state variables of the element, and **.init**, **.handler**, and **.signal** specify the initialization, processing, and signal handler functions of the element. A set of built-in functions are implemented to operate on the input and output ports, as summarized in Table 1.

Similar to Click, ClickNP also uses a simple script to specify a configuration of a network function. The configuration has two parts: *declarations* and *connections*, following the similar syntax of Click language [29]. One thing worth noting is that in ClickNP we can use a keyword **host** to annotate an element, which will cause the element to be compiled into CPU binary and executed on CPU.

Table 1: Built-in operations on ClickNP channels.

uint get_input_port()	Get bitmap of all input ports with available data.
bool test_input_port(uint id)	Test the input port indicated by id.
flit read_input_port(uint id)	Read the input port indicated by id.
flit peek_input_port(uint id)	Peek input data from the port indicated by id.
void set_output_port(uint id, flit x)	Set a flit to the output port. The flit is written to the channel when the handler returns.
ClSignal read_signal()	Read a signal from signal port.
void set_signal(ClSignal p)	Set an output signal on signal port.
return (uint bitmap)	Return value of **.handler** specifies a bitmap of input port(s) to be read on next iteration.

3.2.3 ClickNP tool-chain

The ClickNP tool-chain contains a ClickNP compiler as the front-end, and a C/C++ compiler (*e.g.*, Visual Studio or GCC) and an HLS tool (*e.g.*, Altera OpenCL SDK or Xilinx Vivado HLS) as the back-end. As shown in Figure 2, to write a ClickNP program, a developer needs to divide her code into three parts: (1) A set of elements, each of which implements a conceptually simple operation, (2) A configuration file that specifies the connectivity among these elements, and (3) A host manager that initialize each element and control their behavior during the runtime, *e.g.*, according to the input of administrators. These three parts of source code are fed into the ClickNP compiler and translated into intermediate source files for both host program and FPGA program. The host program can be directly compiled by a normal C/C++ compiler, while the FPGA program is synthesized using commercial HLS tools. Existing commercial HLS tools can determine a maximum clock frequency of each element through timing analysis. Then, the clock of a ClickNP processing graph is constrained by the slowest element in the graph. Additionally, HLS tools may also generate an optimization report which shows the dependency among the operations in an element. An element is *fully pipelined* if all dependency is resolved and the element achieves the optimal throughput by processing one flit in every clock cycle.

4. PARALLELIZING IN FPGA

As discussed in §2.1, it is critical to fully utilize the parallelism inside FPGA in order to speed up processing. ClickNP exploits FPGA parallelism both at element-level and inside an element.

4.1 Parallelism across elements

The modular architecture of ClickNP makes it natural to exploit parallelisms across different elements. The ClickNP tool-chain maps each element into a hardware block in FPGA.

```
1  .element Count <1, 1> {
2    .state{
3      ulong count;
4    }
5    .init{
6      count = 0;
7    }
8    .handler{
9      if (get_input_port() != PORT_1) {
10       return (PORT_1);
11     }
12     flit x;
13     x = read_input_port(PORT_1);
14     if (x.fd.sop) count = count + 1;
15     set_output_port(PORT_1, x);
16
17     return (PORT_1);
18   }
19   .signal{
20     ClSignal p;
21     p.Sig.LParam[0] = count;
22     set_signal(p);
23   }
24 }
```

(a)

```
1 Count :: cnt @
2 Tee :: tee
3 host PktLogger :: logger
4
5 from_tor -> cnt -> tee [1] -> to_tor
6 tee [2] -> logger
```

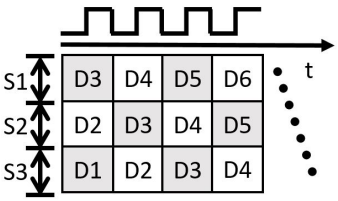
(b)

Figure 4: ClickNP language to write elements and specify configurations. An element annotated with host keyword is compiled and executed on the CPU. An element annotated with "@" is required to receive control signals from the manager thread. "From_tor" and "to_tor" are two built-in elements that represent input and output of an Ethernet port on FPGA. The return value of the handler function specifies a bit-mask of input ports that will be checked in next round.

These logic blocks are interconnected with FIFO buffers, and can work completely in parallel. To this end, one can think of each element in a ClickNP configuration as a tiny, independent core with customized logic. Packets flow from one element to another along a *processing pipeline*. This type of parallelism is called *pipeline parallelism* or *task parallelism*. Furthermore, if a single processing pipeline does not have enough processing power, we can duplicate multiple such pipelines in FPGA and divide data into these pipelines using a load-balancing element, *i.e.*, exploiting *data parallelism*. For network traffic, there are both data parallelism (at packet-level or flow-level) and pipeline parallelism that can be utilized to speed up processing. ClickNP is very flexible and can be configured to capture both types of parallelism with little efforts.

4.2 Parallelism inside element

Unlike CPU, which executes instructions in memory with limited parallelism, FPGA synthesizes operations into hard-



(a)

```
1      r = read_input_port (PORT_1);
2  S1: y = mem[r.x]+1;
3  S2: mem[r.x] = y;
4      set_output_port (PORT_1, y);
```

(b)

```
1      r = read_input_port (PORT_1);
2  P1: if ( r.x == buf_addr ) {
3         y_temp = buf_val;
4      } else {
5         y_temp = mem[r.x];
6      }
7      mem[buf_addr] = buf_val;
8  S1: y = y_temp + 1;
9  S2: buf_addr = r.x;
10     buf_val  = y;
11     set_output_port (PORT_1, y);
```

(c)

Figure 5: Illustration of dependency. (a) No dependency. Sn means a pipeline stage, Dn is a datum. (b) Memory dependency occurs when states are stored in memory and need to be updated. (c) Resolve memory dependency using delayed write.

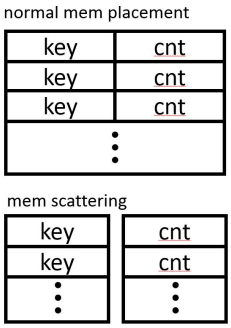
ware logic, and therefore can be evaluated in parallel without instruction load overhead. If a datum requires multiple dependent operations in one handler function, HLS tools will schedule these operations into pipeline stages in a synchronized manner. At every clock, the result of one stage moves to the next stage, and at the same time, a new datum is inputed into this stage, as shown in Figure 5(a). This way, the handler function can process a datum at every clock cycle and achieve maximum throughput. However, in practice, this efficient pipeline processing could break under two situations: (1) there is *memory dependency* among operations; and (2) there are *unbalanced* stages. In the following two subsections, we will discuss these two issues in details and present our solutions.

4.2.1 Minimize memory dependency

If two operations access the same memory location, and at least one of them is *write*, we call these two operations depend on each other [18]. Operations with *memory dependency* cannot be evaluated at the same time, as each memory access has one cycle latency and the semantic correctness of the program strongly depends on the order of operations. As shown in Figure 5(b), **S1** and **S2** depend on each other: **S2** has to be delayed until **S1** finishes, and only after **S2** finishes can **S1** operate on new input data. Therefore, the function will take two cycles to process one datum. Memory dependency can be rather complicated for some processing algorithms, but thanks to the modular architecture of ClickNP,

```
1  struct hash_entry
2  {
3    ulong key;
4    ulong cnt;
5  } A[100];
6
7  .handler {
8    ...
9    idx = hash (h);
10 S1: if (A[idx].key==k)
11   {
12 S2: A[idx].cnt ++;
13   }
14   ...
15 }
```

(a)



(b)

Figure 6: Memory scattering.

most elements perform only simple tasks and the *read-write* memory dependency shown in Figure 5(b) is the most common case we have encountered.

One way to remove this memory dependency is to store data in registers only. Since registers are fast enough to perform read, computation and write back within one cycle, there would be no *read-write* dependency at all. Indeed, compared to CPU, FPGA has a much larger number of registers, *i.e.*, 697Kbit for Altera Stratix V, which can be used whenever possible to reduce memory dependency. The ClickNP compiler aggressively assigns registers to variables as long as all accesses to the variable refer to a constant address – either the variable is a scalar or an array entry with constant offset. Certainly, the programmer can use "register" or "local/global" keywords to explictly instruct the compiler to place a variable (can also be an array) in register, BRAM or onboard DDR memory.

For large data, they have to be stored in memory. Fortunately, we can still use a technique called *delayed write* to resolve the *read-write* memory dependency in Figure 5(b). The core idea is to buffer the new data in a register and delay the write operation until the next read operation. If the next read accesses the same location, it will read the value from the buffer register directly. Otherwise, the read can operate in parallel with the delayed write operation as they are going to access different memory locations[3]. Figure 5(c) shows the code snippet with delayed write. Since there is no longer memory dependency in the code, the element can process a datum in one cycle. By default, ClickNP compiler automatically applies *delayed write* for an array (generating similar code as shown in Figure 5(b)).

One subtle issue will occur when using an array of *struct* variables. Figure 6(a) shows such an example, where a hash table is used to maintain a count for every flow. We find **S2** will have a memory dependency to **S1**, although they are accessing different fields of a **struct**. The reason is that almost all current HLS tools will treat a **struct** array as a single-dimension array with a large bit-width – equal to the size of the **struct**, and use only one arbitrator to control access. We call this type of memory dependency *pseudo dependency*, as physically, the two fields, *key* and *cnt*, can be on different

[3]Most BRAM in FPGA has two ports.

memory locations. To resolve this issue, ClickNP employs a simple technique called *memory scattering*, which automatically translates a **struct** array into several independent arrays, each for a field in the **struct**, and assigns them into different BRAMs (Figure 6(b)). With *memory scattering*, **S1** no longer depends on **S2**. So the pipeline can be inferred by HLS tools, and when **S2** is still operating, a new datum can be clocked in and processed by **S1**. It is worth noting that memory scattering is only applied for elements in FPGA and disabled if elements are assigned to run on the host CPU.

We note that the above techniques may not resolve all memory dependencies. In many cases, it requires programmers to re-factor their code or even change the algorithms to ensure their implementation can be fully pipelined in FPGA.

4.2.2 Balance pipeline stages

Ideally, we require every stage in one processing pipeline to have the same speed. *i.e.*, processing a datum at one clock cycle. However, if the process at each stage is unbalanced and some stages need more cycles than others, these fat stages will limit the whole throughput of the pipeline. For example, in Figure 7(a), **S1** is a loop operation. Since each iteration takes one cycle (**S2**), the whole loop will need N cycles to finish, significantly reducing the pipeline throughput. Figure 7(b) shows another example, which implements a cache in BRAM for a global table (*gmem*) in DDR. Although the "else" branch is seldom hit, it generates a fat stage in the pipeline (taking hundreds of cycles!), and slows down the processing greatly.

ClickNP uses two techniques to balance the stages inside a pipeline. First, we *unroll* the loop whenever possible. When unrolled, the loop operation effectively breaks into a sequence of small operations, each of which can be finished in one cycle. It is worth noting that unrolling a loop will duplicate the operations in the loop body and thus increase area cost. Therefore, it may be only applicable to loops with simple bodies and small number of iterations. In NFs, we find such small loops are rather common, *e.g.* calculating checksums, shifting packet payload and iterating through possible configurations. ClickNP compiler provides the **.unroll** directive to unroll a loop. While many HLS tools support loop unroll for a known number of iterations, ClickNP extends this capability to unroll a loop whose number of iterations is unknown but under an upper bound that is specified by programmers.

Second, if we identify that an element has both heavy and light-weight operations, we try to separate each type of operations in an individual element. For example, to implement a cache as shown in Figure 7(b), we move the slow "else" branch into another element. This way, the fast path and the slow path would be running asynchronously. If cache miss is rare, the overall processing speed is dominated by the fast path. We will return to this point later in §6. Currently, ClickNP compiler cannot automatically perform such separation for programmers.

```
1 .handler {
2    r = read_input_port (PORT_1);
3    ushort *p = (ushort*) &r.fd.data;
4 S1:for (i = 0; i<N; i++) {
5 S2:  sum += p[i];
6    }
7 }
```

(a)

```
1  .handler {
2     r = read_input_port (PORT_1);
3     idx = hash (r.x);
4  S1:if ( cache[idx].key == r.x ) {
5        o = cache[idx].val;
6  S2:} else {
7        o = gmem[r.x];
8        k = cache[idx].key;
9        gmem[k] = cache[idx].val;
10       cache[idx].key = r.x;
11       cache[idx].val = o;
12    }
13    set_output_port (PORT_1, o);
14 }
```

(b)

Figure 7: Unbalanced pipeline stages.

We have built a ClickNP compiler which serves as the front-end of the ClickNP tool-chain (§3.2.3). For the host program, we use Visual C++ as the backend. We further integrate Altera OpenCL SDK (v15.1) [1] and Xilinx Vivado HLS (v2015.4) [9] as the backend for the FPGA program. The ClickNP compiler contains 4,925 lines of C++ code, which parses configuration file and element declarations, performs optimizations in §4, and generates code specific for each commercial HLS tool. When working with Altera OpenCL, each ClickNP element is compiled into a *kernel* and the connections between elements are translated into Altera extended channels. When using Xilinx Vivado HLS, we compile each element into an IP core and use AXI streams to implement connections between elements. An element can also be compiled to CPU binary and the manager thread will create one worker thread for each host element. Each connection between a host and a FPGA element is mapped to a *slot* of the PCIe I/O channel (§5.3).

Our hardware platform is based on Altera Stratix V FPGA with the Catapult shell [40]. The Catapult shell also contains an OpenCL specific runtime, so that the ClickNP role can communicate with the shell through this runtime layer. The FPGA board has a PCIe Gen2 x8 interface, 4GB onboard DDR memory and two 40G Ethernet ports. By the time of writing this paper, we do not get a Xilinx hardware platform. Therefore, the primary system evaluations are based on the Altera platform using ClickNP+OpenCL, and we use the reports generated by Vivado HLS, *e.g.*, frequency and area cost, to understand the performance of ClickNP+Vivado.

5. IMPLEMENTATION

5.1 ClickNP tool-chain and hardware setup

5.2 ClickNP element library

We have implemented a ClickNP element library that contains nearly 100 elements. Part of them (20%) are derived directly from the Click Modular Router, but re-factored us-

Table 2: A selected set of key elements in ClickNP.

Element	Configuration	Optimizations	Performance				Resource (%)	
			Fmax (MHz)	Peak Throughput	Speedup (FPGA/CPU)	Delay (cycles)	LE	BRAM
L4_Parser (A1-5)	N/A	REG	221.93	113.6 Gbps	31.2x / 41.8x	11	0.8%	0.2%
IPChecksum (A1-4)	N/A	UL	226.8	116.1 Gbps	33.1x / 55.1x	18	2.3%	1.3%
NVGRE_Encap (A1,4)	N/A	REG, UL	221.8	113.6 Gbps	35.5x / 42.9x	9	1.5%	0.6%
AES_CTR (A3)	16B block	UL	217.0	27.8 Gbps	79.9x / 255x	70	4.0%	23.1%
SHA1 (A3)	64B block	MS, UL	220.8	113.0 Gbps	157.5x / 83.1x	105	7.9%	6.6%
CuckooHash (A2)	128K entries	MS, UL, DW	209.7	209.7 Mpps	43.6x / 57.5x	38	2.0%	65.5%
HashTCAM (A2)	16 x 1K	MS, UL, DW	207.4	207.4 Mpps	155.9x / 696x	48	18.7%	22.0%
LPM_Tree (A2)	16K entries	MS, UL, DW	221.8	221.8 Mpps	34.5x / 45.2x	181	4.3%	13.2%
FlowCache (A4)	4-way, 16K	MS, DW	105.6	105.6 Mpps	55.8x / 21.5x	27	5.6%	46.9%
SRPrioQueue (A5)	32 Pkts buffer	REG, UL	214.5	214.5 Mpps	150.3x / 28.6x	41	2.6%	0.6%
RateLimiter (A1,5)	16K flows	MS, DW	141.5	141.5 Mpps	7.0x / 65.3x	14	16.9%	14.1%

Optimization method. REG=Using Registers; MS=Memory Scattering; UL=Unroll Loop; DW=Delay Write.
The **Speedup** column compares the performance between the optimized version and our earlier implementation without applying techniques discussed in §4 as well as a CPU implementation.

ing the ClickNP framework. These elements cover a large range of basic operations of NFs, including packet parsing, checksum computing, encap/decap, hash tables, longest prefix matching (LPM), rate limiting, cryptographic, and packet scheduling. Due to the modular architecture of ClickNP, the code size of each element is modest. The mean line-of-code (LoC) of an element is 80 and the most complex element, *PacketBuffer*, has 196 lines of C code. Table 2 presents a selected set of key elements we have implemented in ClickNP. Beside element names, we also mark the demonstration NFs (A1∼A5, discussed in §6) in which the element is used. We have heavily applied the optimization techniques discussed earlier in §4.2 to minimize memory dependency and balance pipeline stages. We summarize the optimization techniques used for each element in the 3rd column. For the top part of Table 2, the element needs to touch every byte of a packet. We show the throughput in Gbps. The elements in the bottom part of the table, however, process only the packet header (metadata). Therefore, it makes more sense to measure the throughput using packet per second. We note that the throughput measured in Table 2 is the maximal throughput that the corresponding element can achieve. When they work in a real NF, other components, *e.g.* the Ethernet port, may be the bottleneck. As a reference, we compare the optimized version to our earlier implementation on FPGA without applying the techniques discussed in §4 as well as a CPU implementation. Clearly, after optimization, all these elements can process packet very efficiently, achieving 7∼157x speedup compared to our naive FPGA implementation, and 21∼696x speedup over a software implementation on one CPU core. This performance gain comes from the ability to utilize the vast parallelism in FPGA. Considering the power footprint of FPGA (∼30W) and CPU (∼5W per core), ClickNP elements are 4∼120x more power efficient than CPU.

We also show the processing latency of each element in Table 2. As we see, this latency is low: The mean is $0.19\mu s$ and the maximum is merely $0.8\mu s$ (LPM_Tree). The last two columns summarize the resource utilization of each element. The utilization is normalized to the capacity of the FPGA chip. We can see most elements use only a small number of logic elements. This is reasonable as most operations on packets are simple. HashTCAM and RateLimiter have moderate usage of LEs because these elements have larger arbitration logic. The BRAM usage, however, heavily relies on configurations of elements. For example, the BRAM usage grows linearly with the number of entries supported in a flow table. Overall, our FPGA chip has sufficient capacity to support a meaningful NF containing a handful elements (§6).

5.3 PCIE I/O channel

As aforementioned, one key property of ClickNP is to support joint CPU/FPGA processing. We enable this by designing a high-throughput, low latency PCIe I/O channel. We extend the OpenCL runtime and add a new I/O channel, which is connected to a PCIe switch in the shell. The PCIe switch will arbitrate the access between the newly added I/O channel and other components in the shell, *e.g.*, DDR memory controller.

We leverage the PCIe slot DMA engine in Catapult shell [40], which divides a PCIe Gen2 x8 link into 64 logical sub-channels, *i.e.*, *slots*. Each slot has a pair of send and receive buffers for DMA operations. Among 64 slots, 33 are reserved by Shell and the runtime library to control kernels and access on-board DDR, one slot is used for passing signals to ClickNP elements. The remaining 30 slots, however, are used for data communications among FPGA and CPU elements. To amortize DMA overhead, we aggressively use *batching*. The maximum message size is limited at 64KB.

In FPGA, a special element, called *CmdHub*, which is generated automatically by the ClickNP compiler, redirects the data from different slots to corresponding FPGA elements using FIFO buffers. *CmdHub* is also used to distribute control signals from the manager thread to FPGA elements. To identify the target element, an element ID is embedded in the signal message, and *CmdHub* can read the ID and for-

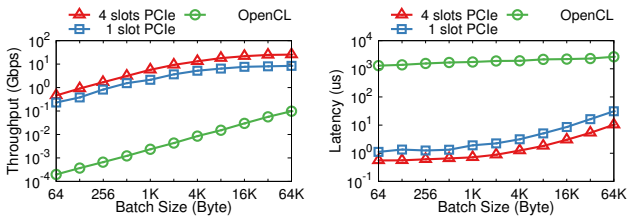


Figure 8: The performance of the PCIe I/O channel. The y-axis is in logarithmic scale.

ward the signal message to the corresponding element, again through FIFO buffers.

Figure 8 shows a benchmark of our PCIe I/O channel with different number of slots and batch sizes. As a base-line, we also measure the performance of OpenCL global memory operations – so far, the only means provided for CPU/FPGA communication in OpenCL [8]. We can see that the maximum throughput of a single slot is around 8.4 Gbps. With 4 slots, the aggregate throughput of the PCIe I/O channel can reach up to 25.6 Gbps. This is the maximum throughput we can get out of our current FPGA chip due to limitation of the clock frequency of the DMA engine. However, the throughput of OpenCL is surprisingly low, less than 1 Gbps. This is because the global memory API is designed to transfer huge amount of data (multiple GB). This may be suitable for applications with large data set, but not for network functions that require strong stream processing capability. Similarly, Figure 8(b) shows the communication latency. Since OpenCL is not optimized for stream processing, the OpenCL latency is as high as 1 ms, usually unacceptable for network functions. In contrast, the PCIe I/O channel has very low latency of 1 μs in polling mode (one core repeatedly polls status register) and 9 μs in interrupt mode (with almost zero CPU overhead).

6. APPLICATIONS

To evaluate the flexibility of ClickNP, we have created five common NFs based on ClickNP. All of them can run in our test-bed processing real-time traffic. Table 3 summarizes the number of elements included in each network function and the total LoC, including all elements specification and the cofiguration files. Our experience also confirms the ClickNP modular architecture greatly improves the code reuse and simplifies the construction of new NFs. As shown in Table 2, there are many chances to reuse one element in many applications, *e.g.*, L4_Parser is used in all five NFs in this paper (A1-5). Each NF may take 1$\sim$2 weeks for one programmer to develop and debug. We find that the ability of joint CPU/FPGA processing would also greatly help debugging, as we can move an element in question to CPU, so that we can easily print logs to track the problem.

A1. Packet generator (PktGen) and capture (PktCap). PktGen can generate various traffic patterns based on different profiles. It can generate different sized flows and schedule them to start at different time, following given distributions. Generated flows can further pass through different traffic shapers to control the flow rates and their burstiness. PktCap simply redirects all packets it receives to *logger* elements, which are usually located in the host. Since a single logger cannot fully utilize the PCIe I/O channel capacity, PktCap has a Receive Side Scaling (RSS) element in FPGA to distribute packets to multiple loggers based on the hash of flow 5-tuple. Since our PCIe channel has less capacity than 40G NIC, we add an *extractor* element that extracts only important fields of a packet (*e.g.* 5-tuple, DSCP and VLAN tag if any) and forwards these fields (total 16B), together with a timestamp (4B) to the logger element across PCIe. PktCap is one example demonstrating the importance of joint CPU/FPGA processing. Compared to FPGA, CPU has more memory for buffering and can easily access other storages, *e.g.*, HDD/SSD drives as in [32], and therefore it makes more sense to run loggers on CPU.

A2. Openflow firewall (OFW). Our Openflow [34] firewall supports both exact- and wildcard-matching of flows. The exact-match table is implemented using Cuckoo Hashing [38] and contains 128K entries that match against flow 5-tuples. The wild-card match is based on TCAM. However, a naive TCAM implementation with 512 104-bit entries takes 65% logic resource of our FPGA. Instead, we use BRAM-based TCAM [27]. BRAM-based TCAM breaks search key into 8-bit keys and use them to address lookup tables, which trades memory for logic area. A BRAM TCAM with 2K entries takes 14% logic and 43% BRAM. Additionally, we design a HashTCAM to leverage the fact that many entries in flow tables share the same bit-masks. HashTCAM divides the table space into a number of smaller hash tables, each of which is associated with a bit-mask. Any incoming packet will first perform an "and" operation before looking up the hash table. Each entry in the table is also associated with a priority. An arbitrator logic is applied to select the matched entry with the highest priority. HashTCAM has better trade-off between capability and area cost. A HashTCAM with 16K flow entries and 16 distinct bit-masks (similar to Broadcom Trident II [10]) takes 19% logic and 22% BRAM. The manager program always tries to group rules based on their bit-masks and places groups with most rules into HashTCAM. The rest rules, which casnnot fit into any groups in HashTCAM, are then put into BRAM-based TCAM.

A3. IPSec gateway (IPSecGW). One issue with software NFs is that the CPU soon becomes a bottleneck when packets require some computation intensive processing, *e.g.*, IPSec [26]. We have built an IPSec datapath that is able to process IPSec packets with AES-256-CTR encryption and SHA-1 authentication. As shown in §5.2, a single AES_CTR element can achieve only 27.8 Gbps throughput. Therefore, we need two AES_CTR elements to run in parallel to achieve line rate. SHA-1, however, is tricky. The SHA-1 divides a packet into smaller data blocks (64B). Although the computation in one data block can be pipelined, there is a dependency between successive blocks inside one IP packet – the computation of the next block cannot start before the previous block has finished! If we process these data blocks sequentially, the throughput would be as low as 1.07 Gbps. Fortunately, we

can leverage parallelism among different packets. While the processing of a data block of the current packet is still going, we feed a data block of a different packet. Since these two data blocks do not have dependency, they can be processed in parallel. To implement this, we design a new element called *reservo* (short for reservation station), which buffers up to 64 packets and schedules independent blocks to SHA-1 element. After the signature of one packet has been computed, the *reservo* element will send it to a next element that appends SHA-1 HMAC to the packet. There is one more tricky thing. Although SHA-1 element has a fixed latency, the overall latency of a packet is different, *i.e.*, proportional to packet size. When multiple packets are scheduled in SHA-1 computation, these packets may be out-of-order, *e.g.*, a smaller packet behind a large packet may finish earlier. To prevent this, we further design a *reorder buffer* element after SHA-1 element that stores the out-of-order packets and restore the original order according to sequence numbers of packets.

A4. L4 load balancer (L4LB). We implement L4LB according to *multiplexer* (MUX) in Ananta [39]. The MUX server basically looks into the packet header and sees if a *direct address* (DIP) has been assigned to the flow. If so, the packet is forwarded to the server indicated by DIP via a NVGRE tunnel. Otherwise, the MUX server will call a local controller to allocate a DIP for the flow. A flow table is used to record the mapping of flows to their DIPs. To handle the large traffic volume in datacenters, it requires the L4LB to support up to 32 million flows in the flow table. Clearly, such a large flow table cannot fit into the BRAM of FPGA, and has to be stored in onboard DDR memory. However, accessing DDR memory is slow. To improve performance, we create a 4-way associative flow cache in BRAM with 16K cache lines. The Least Recently Used (LRU) algorithm is used to replace entries in the flow cache.

In our implementation, an incoming packet first passes a *parser* element which extracts the 5-tuple and sends them to the *flow cache* element. If the flow is not found in the flow cache, the packet's metadata is forwarded to the global flow table, which reads the full table in DDR. If there is still no matching entry, the packet is the first packet of a flow and a request is sent to an *DIPAlloc* element to allocate a DIP for the flow according to load balancing policy. After the DIP is determined, an entry is inserted into the flow table.

After deciding the DIP of a packet, an encapsulation element will retrieve the next-hop information, *e.g.*, IP address and VNET ID, and generate a NVGRE encapsulated packet accordingly. A flow entry would be invalidated if a FIN packet is received, or a timeout occurs before receiving any new packets from the flow.

We put all elements in FPGA except for the *DIPAlloc* element. Since only the first packet of a flow may hit *DIPAlloc* and the allocation policy also could be complex, it is more suitable to run *DIPAlloc* on CPU, being another example of joint CPU-FPGA processing.

A5. pFabric flow scheduler. As the last application, we use ClickNP to implement one recently proposed packet scheduling discipline – pFabric [12]. pFabric scheduling is simple. It keeps only a shallow buffer (32 packets), and always dequeues the packet with the highest priority. When the buffer is full, the packet with the lowest priority is dropped. pFabric is shown to achieve near-optimal flow completion time in datacenters. In the original paper, the authors proposed using a binary comparison tree (BCT) to select the packet with the highest priority. However, while BCT takes only $O(log_2 N)$ cycles to compute the highest priority packet, there is a dependency between successive selection processes. It is because only when the previous selection finishes can we know the highest priority packet, and then the next selection process can be started reliably. This limitation would require the clock frequency to be at least 300MHz to achieve the line rate of 40Gbps, which is not possible for our current FPGA platform. In this paper, we use a different way to implement pFabric scheduler which is much easier to parallelize. The scheme is based on *shift register priority queue* [35]. Entries are kept in a line of K registers in non-increasing priority order. When dequeuing, all entries are shifted right and the head is popped. This takes just 1 cycle. For an enqueue operation, the metadata of a new packet is forwarded to all entries. And now, with each entry, a local comparison can be performed among the packet in the entry, the new packet, and the packet in the neighboring entry. Since all local comparisons can be carried in parallel, the enqueue operation can also finish in 1 cycle. Enqueue and dequeue operations can further be parallelized. Therefore, a packet can be processed in one cycle.

7. EVALUATION

7.1 Testbed and methodology

We evaluate ClickNP in a testbed of 16 Dell R720 servers (§3.1). For each FPGA board, both Ethernet ports are connected to a Top-of-Rack (ToR) Dell S6000 switch [3]. All ClickNP NFs are running on a Windows Server 2012 R2. We compare ClickNP with other state-of-the-art software NFs. For those NFs running on Linux, we use CentOS 7.2 with kernel version 3.10. In our test, we use PktGen to generate testing traffic at different rates with various packet sizes (up to 56.4 Mpps with 64B packets). To measure the NF processing latency, we embed a generation timestamp in every testing packet. When packets pass the NF, they are looped back to a PktCap which is located with PktGen in the same FPGA. Then we can determine the latency by subtracting the generation timestamp from the receiving time of the packet. The delay induced by the PktGen and PktCap was pre-calibrated via direct loop-back (without NFs) and removed from our data.

7.2 Throughput and latency

OpenFlow firewall. In this experiment, we compare OFW with Linux firewall as well as Click+DPDK [17]. For Linux, we use IPSet to handle exact-match rules, while use IPTables for wildcard rules. As a reference, we also include the performance of Dell S6000 switch, which has limited firewall capability and supports 1.7K wild-card rules. It is worth noting

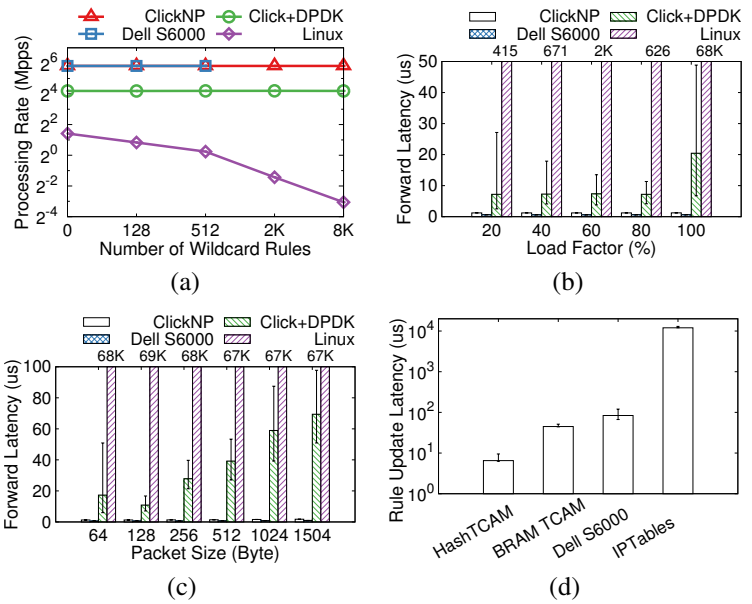


Figure 9: Firewalls. Error bars represents the 5^{th} and 95^{th} percentile. (a) and (b) Packet size is 64B.

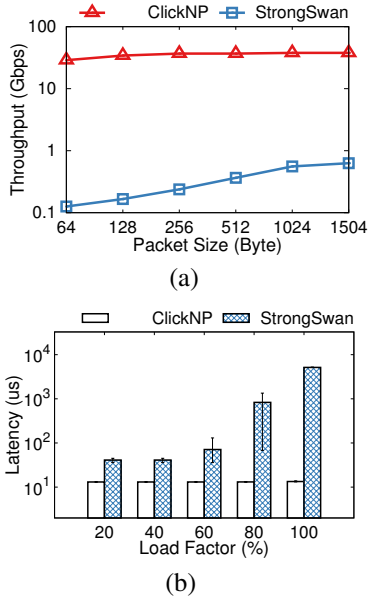


Figure 10: IPSec gateway.

that the original Click+DPDK [17] does not support Receive Side Scaling(RSS). In this work, we have fixed this issue and find when using 4 cores, Click+DPDK already achieves the best performance. But for Linux, we use as many cores as possible (up to 8 due to RSS limitation) for best performance.

Figure 9(a) shows packet processing rates of different firewalls with different number of wild-card rules. The packet size is 64B. We can see that both ClickNP and S6000 can achieve a maximum speed of 56.4 Mpps. Click+DPDK can achieve about 18 Mpps. Since Click uses a static classification tree to implement wildcard-match, the processing speed does not change with the number of rules inserted. Linux IPTables has a low processing speed of 2.67 Mpps, and the speed decreases as the number of rules increases. This is because IPTables performs linear matching for wild-card rules.

Figure 9(b) shows the processing latency under different loads with small packets (64B) and 8K rules. Since each firewall has significantly different capacity, the load factor is normalized to the maximum processing speed of each system. Under all levels of load, FPGA (ClickNP) and ASIC (S6000) solutions have μs-scale latency (1.23μs for ClickNP and 0.62μs for S6000) with very low variance (1.26μs for ClickNP and 0.63μs for S6000 at 95% percentile). However, the software solutions have much larger delay, and also much larger variance. For example, with Click+DPDK, when the load is high, the latency can be as high as $50\mu s$. Figure 9(c) shows the processing latency with different packet sizes and 8K rules. With software solutions, the latency increases with the packet size, mainly due to the larger memory to be copied. In contrast, FPGA and ASIC retain the same latency irrespective to the packet size. In all experiments, the CPU usage of ClickNP OFW is very low ($< 5\%$ of a core).

Finally, Figure 9(d) shows rule insertion latency when there are already 8K rules. Click's static classification tree requires a prior knowledge of all rules, and generating tree for 8K rules takes one minute. IPTables rule insertion takes 12ms, which is proportional to the number of existing rules in the table. Rule insertion in Dell S6000 takes 83.7μs. For ClickNP, inserting a rule into HashTCAM table takes 6.3$\sim$9.5μs for 2$\sim$3 PCIe round-trips, while SRAM TCAM table takes 44.9μs on average to update 13 lookup tables. ClickNP data plane throughput does not degrade during rule insertion. We conclude that OFW has similar performance as ASIC in packet processing, but is flexible and reconfigurable.

IPSec gateway. We compare IPSecGW with StrongSwan [7], using the same cipher suite of AES-256-CTR and SHA1. We setup one IPSec tunnel and Figure 10(a) shows the throughput with different packet sizes. With all sizes, IPSecGW achieves line rates, *i.e.*, 28.8Gbps with 64B packets and 37.8 Gbps with 1500B packets. StrongSwan, however, achieves only a maximum of 628Mbps, and the throughput decreases as packets become smaller. This is because with smaller size, the number of packets needed to be processed increases, and therefore the system needs to compute more SHA1 signatures. Figure 10(b) shows the latency under different load factors. Again, IPSecGW yields constant latency of 13μs, but StrongSwan incurs larger latency with higher variance, up to 5ms!

L4 load balancer. We compare L4LB with Linux Virtual Server (LVS) [4]. To stress test the system, we generate a large number of concurrent UDP flows with 64B packets, targeting one virtual IP (VIP). Figure 11(a) shows the processing rates with different number of concurrent flows. When the number of concurrent flows is less than 8K, L4LB achieves the line rate of 51.2Mpps. However, when the number of concurrent flows becomes larger, the processing rate starts to drop. This is because of flow cache misses in L4LB. When a flow is missing in the flow cache, L4LB has to access the onboard DDR memory, which results in lower per-

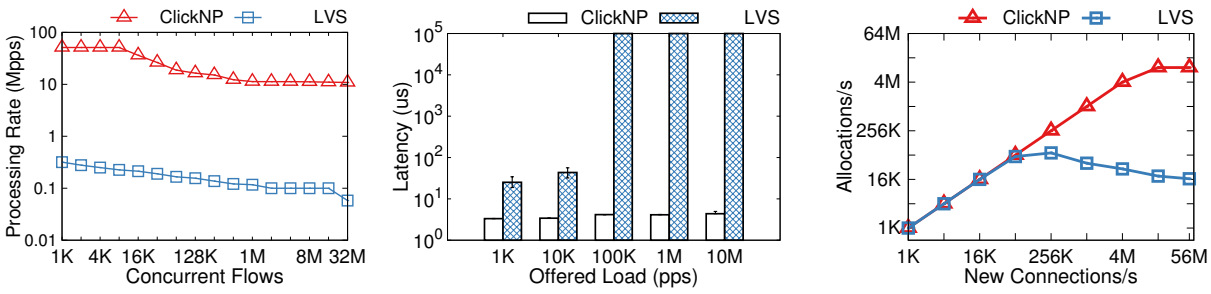


Figure 11: L4 Load Balancer.

formance. When there are too many flows, *e.g.*, 32M, cache miss dominates and for most of the packets, L4LB needs to have one access to the DDR memory. So the processing rate reduces to 11Mpps. In any case, the processing rate of LVS is low. Since LVS associates a VIP to only one CPU core, its processing rate is bound to 200Kpps.

Figure 11(b) shows the latency under different load conditions. In this experiment, we fix the number of concurrent flows to 1 million. We can see that L4LB achieves very low latency of 4μs. LVS, however, incurs around 50μs delay. This delay goes up quickly when the offered load is higher than 100Kpps, which exceeds the capacity of LVS.

Finally, Figure 11(c) compares the capability of L4LB and LVS to accept new flows. In this experiment, we instruct PktGen to generate as many one-packet tiny flows as possible. We can see that L4LB can accept up to 10M new flows per second. Since a single PCIe slot can transfer 16.5M flits per second, the bottleneck is still DDR access. Our *DIPAlloc* element simply allocates DIP in a round-robin manner. For complex allocation algorithms, the CPU core of *DIPAlloc* will be the bottleneck, and the performance can be improved by duplicating *DIPAlloc* elements on more CPU cores. For LVS, due to the limited packet processing capacity, it can only accept up to 75K new flows per second.

7.3 Resource utilization

In this subsection, we evaluate the resource utilization of ClickNP NFs. Table 3 summarizes the results. Except for IPSec gateway which uses most BRAMs to hold coding books, all other NFs only use moderate resources (5∼50%). There is still room to accommodate even more complex NFs.

Next, we study the overhead of fine-grained modularization of ClickNP. Since every element will generate a logic block boundary and use only FIFO buffers to communicate with other blocks, there should be an overhead. To measure this overhead, we create a simple element that only passes data from one input port to an output port. The resource utilization of this *empty* element should well capture the overhead of modularization. Different HLS tools may use different amount of resources, but all are low, with a min of 0.15% to a max of 0.4%. So we conclude ClickNP incurs little overhead due to modularization.

Finally, we want to study the efficiency of RTL code generated by ClickNP, compared to hand-written HDL. To do so, we use NetFPGA [36] as our reference. We extract the key modules in NetFPGA, which are well optimized by experienced Verilog programmers, and implement counterpart elements in ClickNP with the same functionality. We compare the relative area cost between these two implementations using different HLS tools as a backend. The results are summarized in Table 4. Since different tools may have different area costs, we record both the maximum and minimal value. We can see generally, automatically generated HDL code uses more area compared to hand-optimized code. The difference, however, is not very large. For complex modules (shown in the top part of the table), the relative area cost is less than 2x. For tiny modules (shown in the bottom part of the table), the relative area cost appears larger, but the absolute resource usage is small. This is because all HLS tools would generate a fixed overhead that dominates the area cost for tiny modules.

In summary, ClickNP can generate efficient RTL for FPGA that incurs only moderate area cost, which is capable of building practical NFs. Looking forward, FPGA technology is still evolving very rapidly. For example, the next generation FPGA from Altera, Arria 10, would have 2.5x more capacity than the chip we use currently. Therefore, we believe the area cost of HLS would be less of a concern in the future.

Table 3: Summary of ClickNP NFs.

Network Function	LoC†	#Elements	LE	BRAM
Pkt generator	665	6	16%	12%
Pkt capture	250	11	8%	5%
OpenFlow firewall	538	7	32%	54%
IPSec gateway	695	10	35%	74%
L4 load balancer	860	13	36%	38%
pFabric scheduler	584	7	11%	15%

† Total line of code of all element declarations and configuration files.

Table 4: Relative area cost compared to NetFPGA.

NetFPGA Function	LUTs Min / Max	Registers Min / Max	BRAMs Min / Max
Input arbiter	2.1x / 3.4x	1.8x / 2.8x	0.9x / 1.3x
Output queue	1.4x / 2.0x	2.0x / 3.2x	0.9x / 1.2x
Header parser	0.9x / 3.2x	2.1x / 3.2x	N/A
Openflow table	0.9x / 1.6x	1.6x / 2.3x	1.1x / 1.2x
IP checksum	4.3x / 12.1x	9.7x / 32.5x	N/A
Encap	0.9x / 5.2x	1.1x / 10.3x	N/A

7.4 Validation of pFabric

Before we end this section, we show that ClickNP is also a good tool for network research. Thanks to the flexibility and high performance, we can quickly prototype the latest research and apply it to real environments. For example, we can easily implement pFabric scheduler [12] using ClickNP, and apply it in our testbed. In this experiment, we modify a software TCP flow generator [16] to place the flow priority, *i.e.*, the total size of the flow, in packet payload. We generate flows according to the data-mining workload in [12] and further the restrict egress port to be 10 Gbps using a *RateLimit* element. We apply pFabric to schedule flows in egress buffer according to flow priorities. Figure 12 shows the average flow completion time (FCT) of pFabric, TCP with Droptail queue, and the ideal. This experiment validates that pFabric achieves near ideal FCT in this simple scenario.

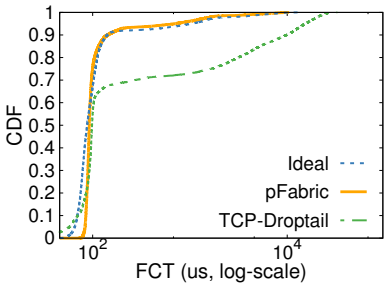


Figure 12: Validation of pFabric.

8. RELATED WORK

Software NFs have great flexibility and scalability. Early studies mainly focus on software-based packet forwarding [20, 21]. They show that multi-core x86 CPU can forward packets at near 10Gbps per server and the capacity can scale by clustering more servers. Recently, many systems have been designed to implement various types of NFs [25, 33, 43]. Similarly, all of these systems exploit the multi-core parallelism in CPUs to achieve close to 10Gbps throughput per machine, and scale out to use more machines when higher capacity is needed. Ananta [39] is a software load-balancer deployed in Microsoft datacenters to provide cloud-scale load-balancing service. While software NFs can scale out to provide more capacity, doing so adds considerable costs in both CAPEX and OPEX [22, 39].

To accelerate software packet processing, previous work has proposed using GPU [26], specialized network processor (NP) [2, 5], and hardware switches [22]. GPU is primarily designed for graphic processing and recently extended to other applications with massive data parallelism. GPU is more suitable for batch operations. Han, *et al.* [26], show that using GPU can achieve 40Gbps packet switching speed. However, batch operations incur high delay. For example, the forwarding latency reported in [26] is about $200\mu s$, two orders of magnitude larger than ClickNP. Compared to GPU, FPGA is more flexible and can be reconfigured to capture data and pipeline parallelisms, both of which are very common in NFs. NPs, however, are specialized to handle network traffic and have many hard-wired network accelerators. In contrast, FPGA is a general computing platform. Beside NFs, FPGA have many other applications in datacenters, making it more attractive to deploy at scale [40]. Hardware switch has limited functionality and its applications are very restricted [22].

FPGA is a mature technology and recently has been deployed to accelerate datacenter services, including NFs [24, 31, 40, 42]. It is well recognized that the programmability of FPGA is low and there is a rich body of literature on improving it, by providing high-level programming abstractions [13–15, 37, 45, 46]. Gorilla [31] proposes a domain-specific high-level language for packet switching on FPGA. Chimpp [42], however, tries to introduce Click model into HDL to develop modular router. ClickNP works along this direction and is complimentary to previous work. ClickNP targets NFs in datacenters, and addresses the programmability issue by providing a highly flexible, modularized architecture and leveraging commercial HLS tools.

The work most related to ours is the Click2FPGA toolchain [41], which compiles an entire Click configuration to FPGA. Its performance, however, is much lower than ClickNP and it lacks support for joint CPU/FPGA packet processing. To the best of our knowledge, ClickNP is the first FPGA-accelerated platform for general NFs processing at 40Gbps line rate, and completely written in high-level language.

9. CONCLUSION

This paper presents ClickNP, an FPGA-accelerated platform for highly flexible and high-performance NFs in commodity servers. ClickNP is completely programmable using high-level language and provides a modular architecture familiar to software programmers in the networking field. ClickNP supports joint CPU/FPGA packet processing and has high performance. Our evaluations show that ClickNP improves the throughput of NFs by 10x compared to state-of-the-art software NFs, while alos reducing latency by 10x. Our work makes a concrete case showing FPGA is capable for accelerating NFs in datacenters. Also, we demonstrate that high-level programming for FPGA is actually feasible and practical. One limitation of FPGA programming, however, is that the compilation time is rather long, *e.g.*, 1∼2 hours, largely due to HDL synthesis tools. ClickNP alleviates this pain with its cross-platform ability, and hopes most bugs could be detected by running elements on CPU. However, in the long term, HDL synthesis tools should be optimized to greatly shorten their compilation time.

Acknowledgements

We would like to thank Andrew Putnam, Derek Chiou, and Doug Burger for all technical discussions. We'd also like to thank the whole Catapult v-team at Microsoft for the Catapult Shell and support of OpenCL programming. We thank Albert Greenberg and Dave Maltz for their support and suggestions on the project. We thank Tong He for his contribution on ClickNP PCIe channel development. Finally, we also thank our shepherd, KyoungSoo Park, and other anonymous reviewers for their valuable feedbacks and comments.

10. REFERENCES

[1] Altera SDK for OpenCL. http://www.altera.com/.

[2] Cavium Networks OCTEON II processors. http://www.caviumnetworks.com.

[3] Dell networking s6000 spec sheet.

[4] Linux virtual server. http://www.linuxvirtualserver.org/.

[5] Netronome Flow Processor NFP-6xxx. https://netronome.com/product/nfp-6xxx/.

[6] SDAccel Development Environment. http://www.xilinx.com/.

[7] Strongswan ipsec-based vpn. https://www.strongswan.org/.

[8] The OpenCL Specifications ver 2.1. Khronos Group.

[9] Vivado Design Suite. http://www.xilinx.com/.

[10] Ethernet switch series, 2013. Broadcom Trident II.

[11] Introducing EDR 100GB/s - Enabling the Use of Data, 2014. Mellanox White Paper.

[12] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, 2013.

[13] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *ACM SIGPLAN Notices*, volume 45, pages 89–108. ACM, 2010.

[14] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Proc. ACM Annual Design Automation Conf.*, 2012.

[15] D. F. Bacon, R. Rabbah, and S. Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[16] W. Bai, L. Chen, K. Chen, and H. Wu. Enabling ecn in multi-service multi-queue data centers. In *Proc. USENIX NSDI*, 2016.

[17] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proc. ANCS*, 2015.

[18] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966.

[19] B. Betkaoui, D. B. Thomas, and W. Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, 2010.

[20] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proc. ACM SOSP*, 2009.

[21] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proc. ACM CoNEXT*, 2008.

[22] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proc. ACM SIGCOMM*, 2014.

[23] A. Greenberg. Windows Azure: Scaling SDN in Public Cloud, 2014. OpenNet Submit.

[24] A. Greenberg. SDN for the Cloud, 2015. Keynote at SIGCOMM 2015 (https://azure.microsoft.com/en-us/blog/microsoft-showcases-software-defined-networking-innovation-at-sigcomm-v2/).

[25] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *ACM SIGCOMM CCR*, 39(2):20–26, Mar. 2009.

[26] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proc. ACM SIGCOMM*, 2010.

[27] W. Jiang. Scalable ternary content addressable memory implementation using fpgas. In *Proc. ANCS*, 2013.

[28] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *IEEE Computer Society Symposium on VLSI*, July 2010.

[29] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[30] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *Proc. USENIX NSDI*, Berkeley, CA, USA, 2014.

[31] M. Lavasani, L. Dennison, and D. Chiou. Compiling high throughput network processors. In *Proc. FPGA*, 2012.

[32] J. Lee, S. Lee, J. Lee, Y. Yi, and K. Park. Flosis: a highly scalable network flow capture system for fast retrieval and storage efficiency. In *Proc. USENIX ATC*, 2015.

[33] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proc. USENIX NSDI*, 2014.

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[35] S.-W. Moon, J. Rexford, and K. G. Shin. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers*, 2000.

[36] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. Netfpga: Reusable router architecture for experimental research. In *Proc. PRESTO*, 2008.

[37] R. S. Nikhil and Arvind. What is bluespec? *ACM SIGDA Newsletter*, 39(1):1–1, Jan. 2009.

[38] R. Pagh and F. F. Rodler. Cuckoo hashing. *Algorithms - ESA 2001. Lecture Notes in Computer Science 2161*, 2001.

[39] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proc. ACM SIGCOMM*, 2013.

[40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[41] T. Rinta-aho, M. Karlstedt, and M. P. Desai. The click2netfpga toolchain. In *Proc. USENIX ATC*, 2012.

[42] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat. Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware. In *Proc. ANCS*, 2010.

[43] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. USENIX NSDI*, 2012.

[44] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback recovery for middleboxes. In *Proc. ACM SIGCOMM*, 2015.

[45] D. Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 2011.

[46] R. Wester. A transformation-based approach to hardware design using higher-order functions. 2015.