Emu: Rapid Prototyping of Networking Services

Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, and Jonny Shipton, *University of Cambridge;* Richard Clegg, *Queen Mary University of London;* Luo Mai, *Imperial College London;* Pietro Bressana and Robert Soule, *Università della Svizzera italiana;* Richard Mortier, *University of Cambridge;* Paolo Costa, *Microsoft Research;* Peter Pietzuch, *Imperial College London;* Jon Crowcroft, Andrew W Moore, and Noa Zilberman, *University of Cambridge*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/sultana

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Emu: Rapid Prototyping of Networking Services

Nik Sultana[†], Salvator Galea[†], David Greaves[†], Marcin Wójcik[†], Jonny Shipton[†],
Richard G. Clegg[‡], Luo Mai[§], Pietro Bressana[*], Robert Soulé[*], Richard Mortier[†],
Paolo Costa[♯], Peter Pietzuch[§], Jon Crowcroft[†], Andrew W. Moore[†], Noa Zilberman[†]

[†]*University of Cambridge,* [‡]*Queen Mary University of London,*
[§]*Imperial College London,* [*]*University of Lugano,* [♯]*Microsoft Research*

Abstract

Due to their performance and flexibility, FPGAs are an attractive platform for the execution of network functions. It has been a challenge for a long time though to make FPGA programming accessible to a large audience of developers. An appealing solution is to compile code from a general-purpose language to hardware using high-level synthesis. Unfortunately, current approaches to implement rich network functionality are insufficient because they lack: (i) libraries with abstractions for common network operations and data structures, (ii) bindings to the underlying "substrate" on the FPGA, and (iii) debugging and profiling support.

This paper describes *Emu*, a new standard library for an FPGA hardware compiler that enables developers to rapidly create and deploy network functionality. Emu allows for high-performance designs without being bound to particular packet processing paradigms. Furthermore, it supports running the same programs on CPUs, in Mininet, and on FPGAs, providing a better development environment that includes advanced debugging capabilities. We demonstrate that network functions implemented using Emu have only negligible resource and performance overheads compared with natively-written hardware versions.

1 Introduction

FPGAs are an attractive platform for implementing network functions. They combine the flexibility of software with the performance and predictability of hardware. Major cloud service providers, such as Microsoft, Baidu, and Amazon, already deploy FPGAs in their data centers to accelerate internal and third-party workloads [36, 40], and implement custom network services [8, 34].

Consequently, there has been significant interest in developing tools and techniques to simplify FPGA programming and making FPGAs accessible to a larger fraction of developers. A common approach is to use *high-level synthesis* (HLS), which allows developers to program FPGAs using a general-purpose language (GPL) such as C, which is then compiled to a hardware description language (HDL), such as Verilog or VHDL.

Unfortunately, while high-level synthesis undoubtedly simplifies FPGA development, HLS alone is not sufficient to implement rich network functionality. Notably, developers who wish to target FPGAs lack three key components. First, they need library support comparable to that in normal software programming, i.e., they need access to re-usable modules and libraries that provide abstractions for common functions and data structures. Second, they require a binding to the underlying "substrate" on the hardware. Unlike CPUs, on an FPGA, there are usually no operating system (OS) and drivers mediating access to hardware. Finally, they need support for fine-grained debugging capabilities, akin to what is available to software developers today.

We present *Emu*, a framework for network functions on FPGAs. Emu builds on the Kiwi compiler [43] that allows computational scientists to program FPGAs with .NET code. The relationship with Emu to .NET/Kiwi is roughly analogous to that of the stdlib to C/GCC—Emu provides the implementation for essential network functionality. Emu and HLS thus result in a powerful substrate for developers to rapidly implement and deploy network functions using a high-level language.

Moreover, Emu virtualizes the hardware context of the network pipeline, allowing developers to write code that is portable across different heterogeneous targets. Our current implementation supports CPUs, simulation environments, and FPGAs. Using Emu, developers can run their network functions as normal processes, using virtual or real NICs, and using network simulators, simplifying debugging and testing. Emu also offers debugging and profiling tools that enable developers to inspect the behavior of the application at runtime.

While simplifying development is important, most network operators are not willing to sacrifice performance for ease-of-development. With Emu, developers can have both: Emu supports designs with different performance metrics such as bandwidth, latency, or operations-per-second.

Using Emu, we have created various prototype implementations of networking services, ranging from an L2 switch to a high-performance Memcached server [17]. Each service is expressed in C#, which can be transformed to host or FPGA instantiations. The FPGA-centered code, created from the C# compiler output and transformed into Verilog executes, for our prototype, on a NetFPGA SUME card [49].

Domain-specific languages such as P4 [5] or ClickNP [26] are too low-level and are designed to support only specific tasks, e.g., packet processing. In contrast, Emu enables the development of a broader set of services, leveraging its support for general-purpose programming.

We compare the performance of Emu against software-only and native Verilog implementations (§5). Our results show that Emu-generated code significantly outperforms software-only versions in terms of latency, latency variance, and throughput, while having a negligible overhead compared to native implementations.

Overall, this paper makes the following contributions:

1. a *"standard library"* for network services, which allows hardware network functions that go beyond header processing to be written in C#. This enables dynamic, conditional processing for network services such as DNS and Memcached. The framework can be customized for different performance metrics, and we illustrate the tradeoffs involved;
2. an *execution environment* that supports running a single codebase over heterogeneous targets, including CPUs, network simulators, and FPGAs; and
3. *debugging support* that translates high-level idioms for debugging, profiling, and monitoring into a low-level language for controlling runtime program state.

Emu and all datasets used in this paper are publicly available [15], and our FPGA designs will be contributed to the NetFPGA community.

2 Motivation

The goal of Emu is to make it easy for software developers with no expertise in hardware languages to quickly develop, test, and deploy network services on an FPGA. Using Emu, application developers can offload network logic to hardware with only modest effort.

The main reason for moving network services from the CPU to FPGAs is increased performance, as demonstrated by existing applications [24, 46, 47]. Moving network functions to hardware also saves CPU cycles, which would otherwise be spent in polling the network interface card (NIC), as typically done in high-performance packet-processing frameworks such as DPDK [52] or netmap [37].

Different data center services, however, have different performance goals. Some applications are throughput-sensitive, e.g., a streaming service, while for others latency is the primary concern [11]. Further, in some cases, latency can be an indirect contributor to low application performance [21]. For example, in Memcached, even tens of microseconds are sufficient to drop the number of queries-per-second significantly [50]. By providing a set of suitable abstractions and APIs, Emu allows developers to optimize towards their preferred performance metric such as ease-of-coding, throughput, or latency.

Our approach can be seen as an example of network paravirtualization: it allows high-level network primitives to be compiled to the paravirtualized hardware (e.g., FPGA or CPU) via the Emu framework. This has the potential to foster innovation at the NIC level, with vendors adding custom logic to natively support some of our high-level APIs. Our library can then be extended to map API calls such as those communicating packets, or doing novel data manipulation (e.g., match-action table processing such as longest-prefix matching, hash and checksum computation, and other conditional operations at line-rate) to custom hardware blocks when available and to rely on paravirtualization, otherwise.

While many consider the translation from a general-purpose language to a hardware language to be the main challenge, there is another important obstacle, namely providing support for debugging an application. Debugging FPGA programs requires the use of hardware-level simulators [32, 42] or probing tools [12], and most network service developers are unfamiliar with these tools. Emu addresses this problem on two levels: (i) it allows application code to be run in a x86 runtime environment. This enables developers to verify and debug the functionality of their code, speeding up the development process; (ii) it provides debugging, monitoring, and profiling tools for the application while running on the hardware. It does this by offering familiar GPL-like abstractions, fitting application developers' capabilities.

Previous work tried to address only a subset of these challenges, as we summarize in Table 1. Past solu-

Solution	What is it?	Target Applications	Processing Paradigm	Language	Performance Metric	Debug Environment[1]	Compiler to Verilog
Emu	"Standard library"	Networking applications	Any	.NET	User defined (see §3.2)	x86, Mininet and Emu env.	Kiwi
Kiwi	Compiler and libraries	Scientific applications	Any	.NET	Execution time/area	x86	Kiwi
Vivado HLS	Compiler and libraries	Scientific applications	Any	C, C++, System C	Throughput	C simulation	Vivado HLS
SDNet	Programming environment	Networking applications	Packet processing	PX/P4	Throughput	C++ simulation	SDNet
P4	Programming language	Networking applications	Packet processing	P4	Throughput	P4 behavioral simulator, Mininet	P4 compiler, then P4FPGA/SDNet.
ClickNP	Programming language/model	Networking applications	Packet processing	ClickNP	Throughput	Undefined	ClickNP, then Altera OpenCL or Vivado HLS

[1]Excluding RTL simulators, accessible on the HDL level to all solutions

Table 1: Comparison between different representative solutions for enabling networking services in hardware

tions either focus on packet processing applications exclusively (e.g., P4 [5] and ClickNP [26]) or target scientific applications only without providing abstractions appropriate for networking (e.g., Kiwi and Vivado HLS). Emu tackles both challenges, while combining additional advantages: supporting a heterogeneous debug environment, as well as user-defined optimizations for performance.

Interestingly, while Vivado HLS specifies latency as a performance metric, this refers to increasing parallelism within the design, rather than network latency metrics. An example showing how increasing parallelism adds to latency is given in §5.3.

3 Emu framework

With Emu, developers can use a general-purpose language to implement high-performance network functions that run on FPGAs. The Emu runtime provides an abstract target environment, and a library of functionality that can execute on both CPUs and FPGAs, simplifying debugging and deployment. Moreover, Emu provides an interface to intellectual property (IP) blocks, i.e., specialized modules that take advantage of hardware features (§3.4). This further abstracts away the details of hardware development. Next, we present an overview of the Emu framework, and describe a typical workflow.

3.1 Background

Emu combines and extends several existing components, including the Kiwi compiler for HLS, and NetFPGA [49] as a hardware target. Note that Emu is not strictly dependent on these specific components—one could use a different HLS compiler or network-attached FPGA.

Kiwi. Originally designed to support scientific computing applications, the Kiwi compiler transforms the target language of .NET compilers, i.e., the common intermediate language (CIL), into a register-transfer level (RTL) description of hardware in Verilog [43]. The Verilog output can then be used to configure FPGAs. We apply Kiwi to the domain of network services and extend it to support networking operations. Emu provides a library to facilitate the development of network functions, and includes some improvements to Kiwi as described in §3.2.

NetFPGA SUME [49] is the latest generation in the NetFPGA family, and provides a low-cost, FPGA-centered PCIe hardware platform for research and education. Alongside several packet-centered reference projects (e.g., an IPv4 router, Ethernet switch, and NIC), NetFPGA has provided the base platform to prototype a variety of high-performance hardware designs for network-centered applications, the best example being prototype hardware for OpenFlow SDN [33].

3.2 Kiwi extensions

Emu provides the following functionality on top of Kiwi: (i) we add support for IP blocks, as defined in §3.4. Although Emu readily generates instances of various components, such as RAMs, ALUs and format converters, we add new support for easily instantiating other IP blocks; (ii) the second extension is needed to mix hard

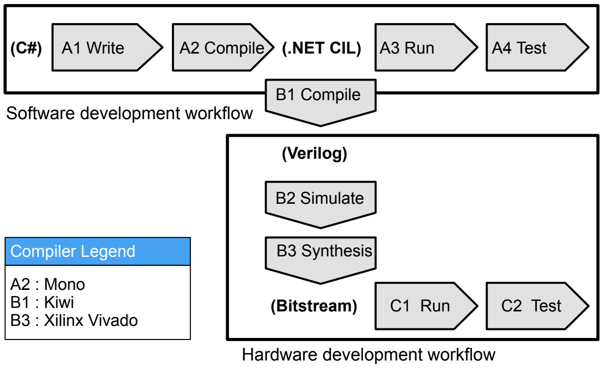

Figure 1: Components of the Emu framework

and soft timing. Kiwi is designed for scientific acceleration, giving it complete freedom over the schedule of operations, which is especially important for multi-cycle floating-point ALU operations. To support the hard timing, cycle-accurate, requirements of network services, Kiwi's scheduler is adapted and paused in parts of the design; (iii) a third extension is the support for casting a byte or a word array into a struct, so that various bit fields take on names and types. C# supports this in the `unsafe` dialect, but the KiwiC version used by Emu only accepts the strongly-typed safe subset of C#; (iv) finally, the largest primitive datatype in C# is the 64-bit word. To achieve higher performance, we require wider I/O busses. Emu defines user types for larger words and provides overloads for all of the arithmetic operators needed.

3.3 Emu overview

Figure 1 shows the main components of the Emu framework, which include: (i) a library tailored to network functions; (ii) runtime support for running C#-coded network programs on a CPU; and (iii) library support for developing and debugging programs. Steps A1, A2, A3 and A4 show the standard C# compilation to bytecode and running/testing on a CPU. B1 uses Kiwi (and Emu extensions) to compile from .NET CIL to Verilog. Steps B2 and B3 (using the NetFPGA framework and the Xilinx compiler) output a bitstream that can be executed on NetFPGA, and this is run and tested in hardware in steps C1 and C2.

Emu extends Kiwi by offering library support customized to the networking domain. Kiwi also provides a "substrate" to support programs that it compiles—the substrate serves as a runtime library for those programs and Emu extends this substrate.

Developers describe a network service in terms of what it does to packets sent and received through network logical ports, which are attached at runtime to network interfaces made available by the OS. The interfaces may be physical or virtual (e.g., a tap device). Emu provides a library and runtime support so developers can quickly test prototypes of network functions written in high-level languages. Layers of abstraction between the .NET runtime and the OS provide virtual/physical network interfaces. By using virtual interfaces, developers can test network functions in a simulator.

```
1  // If the frame does not contain an IPv4 packet then we do not set its output port; this implicitly drops the frame.
2  if (dataplane.tdata.EtherType_Is(EtherTypes.IPv4))
3  {
4    // Configure the metadata such that if we have a hit then set the appropriate output port in the metadata, otherwise broadcast.
5    if (dstmac_lut_hit) {
6      NetFPGA.Set_Output_Port(ref dataplane, lut_element_op);
7    } else {
8      NetFPGA.Broadcast(ref dataplane);
9    }
10 }
11 Kiwi.Pause();
12
13 // Add source MAC to our LUT if it's not already there, thus the switch "learns".
14 if (!srcmac_lut_exist)
15 {
16   LUT[free] = srcmac_port;
17   free = (free > (LUT_SIZE - 1)) ? 0 : free++;
18 }
```

Figure 2: Part of a switch implementation, showing use of our API for protocols (Line 2) and NetFPGA (Line 6)

3.4 Library features

Basic usage. Emu extends the C# code that can be compiled by Kiwi with a library of functions that provide convenience (e.g., by defining frequently-used protocol formats) and performance (e.g, by providing access to carefully crafted IP blocks, see below). Thus any C# code that can be compiled by Kiwi can be used in Emu. An example snippet from our implementation of a switch is provided in Figure 2. Most of the code is standard C#, except for line 11, which controls Kiwi's scheduling (see below), and lines 2 and 6, which use utility functions.

Protocol parsing. Parsers for commonly-used packet formats are available for reuse. As an example, Figure 3 shows the code to instantiate some of the parsers used in the NAT implementation (§4.4). All parsers that may be needed during runtime are instantiated on loading, and, as the snippet shows, it can handle TCP over IPv4 over Ethernet, as well as ARP over Ethernet.

Writing new parsers for custom protocols is straight-

```
var eth = new EthernetWrapper(dataplane.tdata);
var ip = new IPv4Wrapper(dataplane.tdata);
var tcp = new TCPWrapper(dataplane.tdata);
var arp = new ARPWrapper(dataplane.tdata);
```

Figure 3: Parsers for different protocol formats

```
public uint DestinationIPAddress
  { get { return BitUtil.Get32( ips, 0); }
    set { BitUtil.Set32(ref ips, 0, value); } }

public uint SourceIPAddress
  { get { return BitUtil.Get32( ips, 4); }
    set { BitUtil.Set32(ref ips, 4, value); } }
```

Figure 4: Parsing IPv4 headers

forward. Figure 4 shows how two IPv4 fields are manipulated using standard C# programming style as well the utility functions `BitUtil.Get32` and `BitUtil.Set32`.

Using IP blocks. While C# provides an easy development environment, to maximize the performance of a design, it is sometime recommended to use specialized IP blocks that take advantage of the hardware capabilities, such content addressable memory (CAM) used in some of our implementations. These blocks are accessible through the facilities of Kiwi, as mentioned in §3.2.

An example use of an IP block is a hashing module. Figure 5 shows the C# implementation of the protocol required to seed a value (when the hash is used in streaming mode). The protocol involves two signals, `init_hash_ready` and `init_hash_enable`, used for handshaking, and a bundle of eight signals `data_in` used for sending a byte to the core. We can implement the handling of arbitrary protocols in C#, and this enables us to interface with any IP block.

Multi-threading and scheduling control. Kiwi reinterprets concurrency primitives that are used when programming software to improve its hardware generation. It provides a thread-based concurrency library with two type of semantics: (i) software semantics reduces to concurrency primitives provided by .NET, while (ii) hardware semantics forms logical circuits in which parallel threads may be wired into parallel logical sub-circuits.

Using these types of semantics, .NET programs may be executed on general-purpose x86 CPUs by using the software semantics, or on FPGAs by using the logical circuit semantics. In the latter case, Kiwi produces descriptions with much finer parallelism than what is possible on software platforms, whose parallelism is at most instruction-level. We take advantage of this and further refine it to achieve maximal pipelining of projects.

For high performance, developers can also aid Kiwi in scheduling computations across time using annotations,

```
public static void Seed(byte data_in)
 {
   while (init_hash_ready) { Kiwi.Pause(); }
   PearsonHash.data_in = data_in;
   init_hash_enable = true;
   Kiwi.Pause();
   while (!init_hash_ready) { Kiwi.Pause(); }
   Kiwi.Pause();
   init_hash_enable = false;
   Kiwi.Pause();
 }
```

Figure 5: Part of the wrapper for our hashing module

```
// Extract the frame from NetFPGA_Data into a byte array.
public static void Get_Frame (NetFPGA_Data src, ref byte[]
      dst)
...

// Move the contents of a byte array into the frame field
      in NetFPGA_Data.
public static void Set_Frame (byte[] src, ref NetFPGA_Data
      dst)
...

// Read the input port (i.e., port on which we received the
       frame).
public static uint Read_Input_Port (NetFPGA_Data dataplane)
...

// Set the output port to a specific value. (i.e., the port
       to which we are forwarding the frame.)
public static void Set_Output_Port (ref NetFPGA_Data
      dataplane, ulong value)
...
```

Figure 6: Utility functions for interacting with the FPGA dataplane

as shown in line 11 in Figure 2. This breaks up computation and allows Kiwi to schedule a suitable amount of computation in a single clock cycle by providing a cycle-accurate notion where needed. If Kiwi schedules too little computation, it is inefficient; if it schedules too much computation, the implementation on the target FPGA device fails. Currently, Kiwi is target oblivious, i.e., it does not have information about clock rates.

Utility functions. In addition to the purpose-specific APIs described in previous sections, Emu also includes general utility functions. These form a library of C# code and are intended to help abstract unnecessary details, such as the functions listed in Figure 6 for interacting with the FPGA target. One could have different sets of such functions for different targets, e.g., without changing the code for protocol parsing or IP blocks.

3.5 Debug-related features

Emu produces a debug environment by the systematic extension of programs to interpret *direction commands* at runtime to enable debugging, monitoring and profil-

```
if V_trace_idx < max_trace_idx then
  V_trace_buf[V_trace_idx] := V;
  inc V_trace_idx;
  continue
else
  inc V_trace_overflow;
  break
```

Figure 7: Code that implements the direction command "trace *X* max_trace_idx" (If the buffer is not full, the new value of X is logged, the index incremented, and control is returned to the program that hosts this code; otherwise, it indicates depletion of the associated buffer resource and break the program's execution.)

ing. This design came about after we found ourselves extending our ad hoc debugging and monitoring code to support additional features. It is inspired by the commands found in profilers and debuggers such as gdb.

Emu uses a language of direction commands [44]. Figure 7 describes one such command, "trace *V* max_trace_idx", and shows how to express this high-level direction command as a program executable by a controller, with which Emu programs are extended (see Figure 8).

Table 2 lists other supported high-level direction commands. Commands are translated into programs that execute on a simple controller embedded in the program. We model the controller as a *counters, arrays, and stored procedures* (CASP) machine, which refers to the constituents of the machine's memory.

Extending a C# program to support direction commands involves inserting (i) named *extension points* with runtime-modifiable code in a computationally weak language (no recursion); and (ii) *state* used for bookkeeping by that code to implement direction features.

Debugging can also be conducted using *direction packets*. Direction packets are network packets in a custom and simple packet format, whose payload consists of (i) code to be executed by the controller; or (ii) status replies from the controller to the director. It enables us to remotely direct a running program, similar to gdb's "remote serial protocol" [18].

Emu minimizes the overhead that these features introduce at runtime by extending a program (before compilation) to support the precise set of required debugging or profiling features. This frugality does not come at the cost of inflexibility, however, because the extension points at runtime can be reconfigured to perform different debugging or profiling functions.

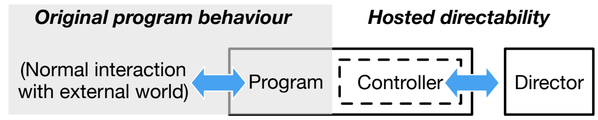

Figure 8: *Controller* embedded into the program, acting as the agent of the *director* (The director and controller exchange commands and their outputs.)

3.6 Limitations

The main limitation of Emu when compared to HDLs is the lack of low-level control over hardware designs, and here Emu is partly limited by Kiwi's capabilities. Kiwi does not yet allow one to internalize instances of an HDL module, and this forces Emu to interface with such modules instead of instantiating them.

In addition, Emu currently supports only a limited number of protocols, but developers can extend the library to support more protocols (see Figure 4).

Finally, depending on the required performance, developers must be aware of the hardware that the design is deployed on, or is interfacing with. For example, for a given throughput, a wider I/O bus may be required.

4 Use cases

We have implemented different networking services to demonstrate the benefits of Emu. These include forwarding (§4.1), measurement and monitoring (§4.2), performance sensitive applications (§4.3), and more complex applications such as NAT and caching (§4.4). The use cases cover a range of network services, and can include bespoke features, e.g., encryption schemes. The use case implementations are available under an open-source license [15]. Some of the applications are also available as contributed projects to the NetFPGA community, starting with NetFPGA SUME release 1.4.0.

4.1 Packet forwarding

Learning switch. We implement a standard layer-2 learning switch, similar in functionality to the NetFPGA SUME reference switch [45]. Beyond header processing, which is a basic networking function, it provides an example of how content addressable memory (CAM) is implemented in Emu, and how a native FPGA IP CAM block can be used. While the first option does not burden developers with implementation details, the latter provides better resource usage and timing performance. A simplified version of our implementation is shown in Figure 2. The full version is around 150 lines of C#, and the resulting Verilog is around 500 lines.

Command		Behaviour
print X		Print the value of variable X from the source program.
break L $\langle B\rangle$		Activate a (conditional) breakpoint at the position of label L.
unbreak L		Deactivate a breakpoint.
backtrace $\langle\$\rangle$		Print the "function call stack".
watch X $\langle B\rangle$		Break when X is updated and satisfies a given condition.
unwatch X		Cancel the effect of the "watch" command.
count	reads X $\langle B\rangle$ $\langle\$\rangle$ writes X $\langle B\rangle$ $\langle\$\rangle$ calls *fname* $\langle B\rangle$ $\langle\$\rangle$	Count the reads or writes to a variable X, or the calls to a function *fname*.
trace	start X $\langle B\rangle$ $\langle\$\rangle$	Trace a variable, subject to a satisfied condition, and up to some length.
	stop X	Stop tracing a variable.
	clear X	Clear a variable's trace buffer.
	print X	Print the contents of a variable's trace buffer.
	full X	Check if a variable's trace buffer is full.

Table 2: Directing commands (Note that count has similar subcommands to those of trace.)

L3–L4 filter. We provide a tool that emulates the command-line parameter interface of IP tables [35]. Instead of modifying a Linux server's filters, it generates code that slots into our learning switch. This turns the switch into a L3 filter over sets of IP addresses or protocols (ICMP, UDP, and TCP), or an L4 filter over ranges of TCP or UDP ports.

4.2 Measurement and monitoring

ICMP echo. We have implemented an ICMP echo server to obtain two baselines: (i) a *qualitative* baseline on the difficulty of implementing a simple network server, and (ii) a *quantitative* baseline on how much time is saved by avoiding the system bus, CPU, OS, and network stack.

TCP ping. Sometimes the network handles ICMP traffic differently to the protocols used by applications such as TCP and HTTP. For example, a faulty configuration of the network may discard packets on some TCP ports on a machine, but without affecting the reachability of that machine through ICMP [22]. TCP ping involves a simple reachability test by using the first two steps of the three-way connection setup handshake. It is thus a more complex extension of ICMP echo. Our implementation is around 700 lines of C#, and the resulting Verilog is around 1,200 lines.

4.3 Performance-sensitive applications

DNS. We provide a simple DNS server that supports non-recursive queries. Our prototype supports resolution queries from names (of length at most 26 bytes) to IPv4 addresses, but these constraints can be relaxed to handle longer names and IPv6. If the queried name is absent from the resolution table, the server informs the client that it cannot resolve the name. Our implementation is around 700 lines of C#, and the resulting Verilog around 1,200 lines. [1]

Memcached [17] is a well-known distributed in-memory key/value store that caches read results in memory to quickly respond to queries. Its protocol uses a number of basic commands such as `GET` (to retrieve a value associated with the provided key), `SET` (to store a key/value pair) and `DELETE` (to remove a key/value pair), and supports both ASCII and binary protocols.

Memcached is sensitive to latency, and even an extra 20 μs are enough to lose 25% throughput [50]. Our initial Memcached implementation with Emu focussed on latency only and therefore supported only a limited version of the protocol, allowing only `GET`/`SET`/`DELETE` using the binary protocol over UDP, with 6-byte keys and 8-byte values. We later experimented with different extensions of this design, adding support for the ASCII protocol, larger key/value sizes, and for the use of DRAM and multiple CPU cores. These features introduce different trade-offs with respect to latency, throughput, and functionality.

4.4 Other applications

NAT. We provide a network address translation (NAT) service, supporting both UDP and TCP, which was im-

[1] It is a coincidence that the code length is the same as for the TCP ping use case.

```
public class Data {
  public bool matched = false;
  public ulong result = 0;
}
public class LRU {
  public static Data Lookup(ulong key_in) {
    Data res = new Data();
    ulong idx = HashCAM.Read(key_in);
    if (HashCAM.matched) {
      res.matched = HashCAM.matched;
      res.result = NaughtyQ.Read(idx);
      NaughtyQ.BackOfQ(idx);
    }
    return res;
  }
  public static void Cache(ulong key_in, ulong value_in) {
    ulong idx = NaughtyQ.Enlist(value_in);
    HashCAM.Write(key_in, idx);
  }
}
```

Figure 9: Least-recently-used (LRU) cache in Emu

plemented by a second-year undergraduate student. The implementation is written entirely in C#, without the use of Verilog-based cores, and has less than 1,000 lines.

One of the advantages of Emu is that the same code can run on multiple platforms, enabling a better development cycle. We use the NAT service as a test case, compiling it to three different targets: software, Mininet [31], and hardware.

Caching. One potential application that can benefit from offloading to hardware is caching. For example, SwitchKV [27] uses SDN-enabled switches to dynamically route read requests to a cache if content is available. This idea can be extended to directly implement a cache in the data plane, reducing load on storage servers. Implementing a cache in a DSL such as P4, however, would be difficult, because the eviction logic must be managed by the control plane. In contrast, with Emu, one can easily implement a look-aside, least-recently-used (LRU) cache in a few lines, as shown in Figure 9.

5 Evaluation

Our evaluation of Emu has the following aims: (a) provide evidence that using Emu is beneficial in terms of resources and performance, compared with other solutions; and (b) explore if Emu can be used to implement high-performance network services.

5.1 FPGA hardware

At the core of the NetFPGA SUME board is a Xilinx Virtex-7 690T FPGA device. The memory subsystem combines both static random access memory (SRAM) and dynamic random access memory (DRAM). It supports up to 32 GB of RAM, and can run as a stand alone computing unit [23]. NetFPGA SUME's native frequency is 200 MHz.

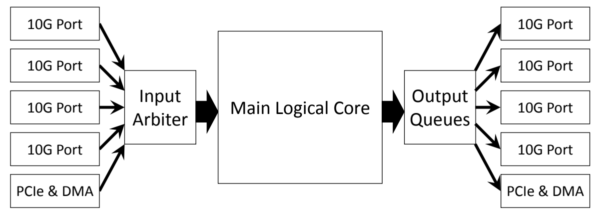

Figure 10: NetFPGA reference pipeline (The input arbiter, logical core, and output queues form the data plane.)

The NetFPGA reference designs share a generic FPGA architecture, shown in Figure 10, with multiple physical interfaces surrounding a logical data-path. Emu capitalizes on this generic NetFPGA design: we target only the main logical core and build upon all other components to be shared between services, thus requiring no hardware expertise.

5.2 Experimental setup

Our experiments are conducted using a server with a single 3.5 GHz Intel Xeon E5-2637 v4 CPU with 64 GB DDR4 memory and a SuperMicro X10-DRG-Q motherboard. The machine runs Ubuntu Linux 14.04 LTS with the kernel version 3.13.0-106-generic. It has a dual port 10 GbE NIC (Intel 82599ES). The machine also includes a NetFPGA SUME board for the performance comparison. We use an Endace DAG 9.2X2 card for accurate latency measurements. All traffic is captured by the DAG card and used to measure the latency of the device-under-test (DUT) alone. The latency of the setup itself is measured first and deducted from all subsequent measurements. For latency measurements, the server runs the service pinned to a single CPU core with a warm cache.

For our throughput measurements, we use the Open Source Network Tester (OSNT) [1] as the traffic source. OSNT replays real traffic traces while modifying traffic rate to find the maximum throughput (e.g. queries per second). When testing, the server is configured to achieve maximum throughput (e.g. using multiple CPU cores), and this configuration changes between tests.

5.3 Comparison against hardware services

Next, we evaluate the immediate overheads of using Emu and show that the resulting implementations are comparable with native HDL designs.

	Emu	NetFPGA reference	P4FPGA
Logic resources	3509	2836	24161
Memory resources	118	87	236
Module latency	8 cycles	6 cycles	85 cycles
Throughput (Mpps)	59.52	59.52	53

Table 3: Comparison between Emu switch (C#), NetFPGA reference switch (Verilog), and P4FPGA switch (P4), using 64 byte packets

We compare the Emu learning switch, written in C# and compiled using Kiwi, with the NetFPGA SUME reference switch written directly in Verilog. We further extend this comparison to a similar design, written in P4 and compiled to NetFPGA SUME [47]. We do not compare with SDNet [39], as done by Dang et al. [10], because P4FPGA has better reported performance. As previous work [47], we use 256-entry tables.

Table 3 shows the resources consumed by the main logical core in each design. These results confirm that the resource overhead is minimal, making Emu an attractive solution. Furthermore, out of the reported resources consumed by Emu core, 85% are used by the CAM, which is an IP block, and only 15% by the C# generated logic. We note that, in all our use cases, the FPGA resources are never exhausted, and consume less than 33% of the logic resources, including the debug controller.

In terms of latency, Emu has only a minor overhead over the main logical core in the NetFPGA SUME reference switch design. In comparison to P4FPGA, Emu provides much lower latency than the compared design, mostly because Emu is not bounded by the match/action paradigm. In terms of throughput, instead, while P4FPGA achieves 53 Mpps for 64 byte packets using a 250 MHz clock, and a header parser for every port, Emu achieves full line rate (59.52 Mpps) using a 200 MHz clock and a single header parser.

Unfortunately, the authors of ClickNP [26] do not provide enough information, such as the FPGA clock rate, which would allow for a fair comparison with Emu. However, their reported packet-processing rate for similar applications (e.g., a firewall with 56 Mpps) is on par with Emu, as is the latency (e.g., 11 cycles for L4 Parser). In terms of resource usage, ClickNP has a resource utilization of 0.9× compared with the NetFPGA reference design's header parser (resp. 3.2× for a multi-threaded design). Emu's resource utilization, instead, is 0.7× with a single-thread design (1.2× with a multi-thread design).

5.4 Comparison against software services

In the previous section, we compared against equivalent implementations running on FPGAs. Now, we explore the performance of the different use cases from §4 against software-based, Linux native counterparts.

Setup. ICMP Echo and TCP Ping are used to evaluate the performance of a simple networking operation. We measure the round-trip time (RTT) required to reply to a request of the DUT alone. Latency measurements are performed for 100K packets. We configure NAT as a gateway to/from the local network, and measure the latency between an input interface from the external network and an output interface to the local one.

The Memcached evaluation uses the memaslap benchmark [30], configured to use a mix of 90% `GET` and 10% `SET` requests with random keys. The Emu Memcached implementation uses UDP and the ASCII protocol. We compare against a Linux Memcached server with 4 threads and 64 MB of memory, also running the UDP and ASCII protocols.

Results. We show the latency and throughput results in Table 4. Across all use cases, Emu achieves a reduction in latency from one to three orders of magnitude. Most importantly, unlike the host-based implementations, Emu's services exhibit a very short tail latency. This is particularly important as in distributed applications the application performance is often bound by the tail latency [11]. This means that not only Emu yields very low latency but it also guarantees *predictable* performance. In contrast, host-based implementations suffer from unpredictable delays and interrupts across the stack and exhibit a much higher variability with the tail-to-average ratio variying from 1.09 to 2.98 (resp. from 1.02 to 1.04 for Emu).

Emu also significantly outperforms host-based solution in term of throughput with improvements ranging from a factor of 2.1 up to a factor of 5.2. Interestingly, these results were obtained using a single-threaded Emu's configuration and could be further improved by instantiating multiple Emu cores. For example, in the Memcached usecase, using four Emu cores (one per port) further increases by 3.7× when considering a workload of 90% `GET` and 10% `SET` requests. SET requests must be applied to all instances, thus their relative ratio in performance cannot improve. The downside is that such an approach requires changes to the main logical core wrapper in NetFPGA SUME.

Optimizations. Further extending the above use cases can be done in different ways. For Memcached, it is possible to increase the memory available to Emu, using ei-

Network service	Emu			Host		
	Average latency (μs)	**99th-perc. latency** (μs)	**Throughput** (million queries/s)	**Average latency** (μs)	**99th-perc. latency** (μs)	**Throughput** (million queries/s)
ICMP Echo	1.09	1.11	3.226	12.28	22.63	1.068
TCP Ping	1.27	1.29	2.105	21.79	65.00	1.012
DNS	1.82	1.86	1.176	126.46	138.33	0.226
NAT	1.32	1.34	2.439	2444.76	6185.27	1.037
Memcached	1.21	1.26	1.932	24.29	28.65	0.876

Table 4: Comparison between services running on a host and Emu-based services (C#)

ther on-board or on-chip memory. On-board memory, e.g., using the DDR3 DRAM memory modules on NetFPGA SUME, has a size advantage, but the disadvantage of increased and variable latency (e.g., due to DRAM refreshes); on-chip memory has the benefit of low, constant latency, but is of smaller size. While NetFPGA SUME has 51 MB of on-chip memory, devices such as Xilinx Ultrascale+ have up to 65 Gbit on-chip, providing a solution at much larger scale. Further scaling can be achieved by using the Emu-based design as a (large) L1 cache, bounded to a few GBs, where cache misses are sent to a host [46] and implemented using the NetFPGA.

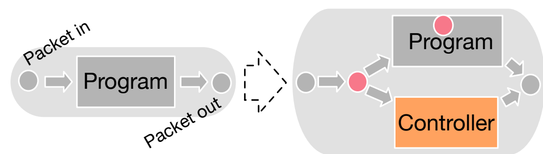

Figure 11: Transformation of the program to include a controller (Normal packets are handled without change, but *direction packet* are passed to the controller. Pink dots represent *extension points*, one of which is added within the control flow of the original program.)

5.5 Debugging

We extend the DNS and Memcached use cases in two ways: (i) adding code to check if a received packet is a *direction packet* intended for the controller (see Figure 11), in which case the controller (and not the original program) processes the packet; (ii) adding an extension point in the body of the (DNS or Memcached) main loop, allowing us to influence and observe the program from that point onwards. We form an enumerated type that corresponds to the program variables whose values the controller may access and change. The code for each value of the enumerated type refers to the program value, e.g., instructing the controller to increment it.

We evaluate Emu's debug environment by carrying out a quantitative analysis of the impact that the controller has on the program in which it is embedded. We measure this impact in terms of *utilization* of resources on the FPGA and the *performance* of the host program.

Table 5 shows the utilization and performance for DNS and Memcached, respectively, extended with different controller features: reading, writing, and incrementing a variable. The impact on utilization and performance is small, and dominated by the controller logic, rather than specific-purpose and runtime-programmable registers. Utilization improvements are due to the optimization process during the place-and-route state in hardware generation; occasionally this results in more utilization-efficient allocations.

An example of using directed packets is the debug process of our Memcached implementation. The Memcached service running on hardware replied with an error message, while no problem was detected in simulation. Using directed packets, we examined the Memcached service: directing the packets to report the checksum calculated within Emu revealed a bug in the checksum implementation and simulation environment.

5.6 Summary

Our evaluation demonstrates the advantages of Emu: (i) hardware resource usage is significantly lower than that of other approaches, adding only modest overhead when compared with bespoke HDL-only designs; (ii) the latency overhead is small compared to HDL designs and is similar to or better than that of other baselines; (iii) the overhead from the debug extensions is negligible, making Emu an attractive debug environment.

Our results also show an important advantage of Emu over host-based solutions: while absolute performance always depends on the CPU cores, memory bandwidth and frequency, FPGAs enjoy the benefit of predictability.

Artefact	Utilization (%)	Performance (%)	
	Logic	Latency	Queries-per-sec
DNS	100.0	100.0	100.0
+**R**	103.4	100.0	100.0
+**W**	115.1	99.5	100.0
+**I**	109.8	99.5	100.0
Memcached	100.0	100.0	100.0
+**R**	99.2	100.0	100.0
+**W**	99.8	100.5	100.0
+**I**	100.6	100.0	100.0

Table 5: Profile of utilization and performance (**R**ead, **W**rite, and **I**ncrement are instructions supported by the controller. Latency is compared at the 99^{th} percentile.)

The median latency of our designs is both 10× lower than the median of the host-based solutions, with a small variance. While the difference between the median and 99^{th} percentile is less than 200 ns for Emu, for host-based designs the variance is in the order of microseconds to tens of microseconds. This not only improves RTT and flow-completion times, but it also enables users to better schedule resources as they know when a reply is due.

6 Related work

FPGAs are increasingly deployed inside data centers, and their performance is getting closer to specialized hardware [51]. Recently there has been a large body of work on how to offload critical network and application services to FPGAs [2, 13, 14, 16, 24, 25, 36, 41, 48]. All of these proposals, however, leverage HDLs, making them unsuitable for the majority of developers who lack hardware skills. Emu addresses this issue by removing most of the challenges related to hardware programming and making FPGAs accessible to non-hardware experts, while retaining high performance.

We are not the first to target this goal and in the past there have been many efforts to make programming FPGAs easier, e.g., using a DSL [6, 7, 9, 38, 39], including network-specific ones [3, 5, 26]. These DSLs typically have a narrow scope and limit the performance or ability to implement certain network services. For example, P4 [5] is a popular DSL for packet processing that supports compilation to hardware including FPGAs. However, it is only applicable to tasks that can be processed by parse-match-action style systems. LINQits [9] proposes a tool chain that compiles an embedded query language (LINQ) into various platforms, including FPGAs, but it is specialized for answering queries and would require considerable adaptation to perform networking tasks. In contrast, Emu does not restrict the set of network services that can be implemented and offers a more general programming environment.

High-level synthesis (HLS) tools [28] generate HDL from high-level languages such as Scala, or Java (using Lime [4]), but they do not offer specific support for network programming. One exception is the Maxeler MPC-N system [29], which provides a "dataflow engine" to offload network computations to hardware. The engine runs kernels that are programmed using a subset of Java, and proprietary tooling. This approach, however, targets a proprietary hardware platform and lacks the ability to run seamlessly on both CPU and FPGAs. Conversely, Emu makes few assumptions about the underlying hardware and can be ported to different FPGAs. In addition, Emu's support for executing programs on a CPU and in simulation, combined with its advanced monitoring and profiling capabilities, greatly simplifies debugging of network programs.

The work in this paper is based on Kiwi [20, 43]. In previous work, Kiwi was used to distribute an application across network-connected hosts [19], but the network-related code was simple and had to be written from the ground up, because it lacked the "standard library" abstractions and debugging support provided by Emu.

7 Conclusion

Although the performance and availability of programmable network hardware has increased, making effective use of it remains beyond the reach of most developers. We have presented Emu, a framework that enables application developers to write network services in a high-level language (C#) and have them automatically compiled to execute across a number of platforms, including traditional CPUs (x86), simulation environments (Mininet), and an FPGA platform (NetFPGA), without compromising on performance.

We showed that the performance of Emu-based network services exceeds software-based solutions and is on par with native HDL implementations. Implementations on Emu permits services to run on different targets, support better debug capabilities and allow for easier transition of workloads among targets.

Acknowledgements. We thank Gordon Brebner, Han Wang, Matthew P. Grosvenor, the anonymous reviewers, and our shepherd, Christopher Rossbach. We acknowledge the support from the UK Engineering and Physical Sciences Research Council (EPSRC) (EP/K032968/1, EP/K034723/1, EP/K031724/2, and an UROP grant), Leverhulme Trust (ECF-2016-289) and Newton Trust, EU H2020 SSICLOPS (644866), SNF (166132) and a Google Faculty Research Award.

References

[1] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, Andrew W. Moore, and Philippe Owezarski. OSNT: Open source network tester. *IEEE Network*, 28(5):6–12, 2014.

[2] Shadi Atalla, Andrea Bianco, Robert Birke, and Lucado Giraudo. NetFPGA-based load balancer for a multi-stage router architecture. In *World Congress on Computer Applications and Information Systems*, pages 1–6. IEEE, Jan 2014.

[3] Michael Attig and Gordon Brebner. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Symposium on Architectures for Networking and Communications Systems*, pages 12–23. IEEE Computer Society, 2011.

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Object Oriented Programming Systems Languages and Applications*, pages 89–108. ACM, 2010.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] Gordon Brebner and Weirong Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014.

[7] Kevin J Brown, Arvind K Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques*, pages 89–100. IEEE, Oct 2011.

[8] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *International Symposium on Microarchitecture*. IEEE, Oct 2016.

[9] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. *SIGARCH Comput. Archit. News*, 41(3):261–272, June 2013.

[10] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki-Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. Network hardware-accelerated consensus. *CoRR*, abs/1605.05619, 2016. URL: `http://arxiv.org/abs/1605.05619`.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[12] Vivado Hardware Debug. `https://www.xilinx.com/products/design-tools/vivado/debug.html`.

[13] Ken Eguro. Automated Dynamic Reconfiguration for High-Performance Regular Expression Searching. In *International Conference on Field-Programmable Technology*. IEEE, Dec 2009.

[14] Ken Eguro and Ramarathnam Venkatesan. FPGAs for trusted cloud computing. In *International Conference on Field-Programmable Logic and Applications*. IEEE, Aug 2012.

[15] Emu Project. `http://www.cl.cam.ac.uk/research/srg/netos/projects/emu/`.

[16] Felix Engelmann, Thomas Lukaseder, Benjamin Erb, Rens van der Heijden, and Frank Kargl. Dynamic packet-filtering in high-speed networks using NetFPGAs. In *Int. Conf. on Future Generation Communication Technologies*, pages 55–59. IEEE, Aug 2014.

[17] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004.

[18] GDB Remote Serial Protocol. `http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html`.

[19] David Greaves and Satnam Singh. Distributing C# methods and threads over Ethernet-connected FPGAs using Kiwi. In *International Conference on Formal Methods and Models for Codesign*, pages 1–9. IEEE, July 2011.

[20] David J. Greaves and Satnam Singh. Designing application specific circuits with concurrent C# programs. In *Formal Methods and Models for Codesign*, pages 21–30. IEEE, July 2010.

[21] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *Symposium on Networked Systems Design and Implementation*, pages 1–14. USENIX Association, 2015.

[22] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. *SIGCOMM Computer Communication Review*, 45(4):139–152, August 2015.

[23] Jong Hun Han, Noa Zilberman, Bjoern A. Zeeb, Andreas Fiessler, and Andrew W. Moore. Prototyping RISC based, reconfigurable networking applications in open source. *CoRR*, abs/1612.05547, 2016. URL: `http://arxiv.org/abs/1612.05547`.

[24] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Symposium on Networked Systems Design and Implementation*, pages 425–438. USENIX Association, 2016.

[25] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Architectural Support for Programming Languages and Operating Systems*, pages 67–81. ACM, 2016.

[26] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, pages 1–14. ACM, 2016.

[27] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Symposium on Networked Systems Design and Implementation*, pages 31–44. USENIX Association, March 2016.

[28] Grant Martin and Gary Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[29] Maxeler MPC-N Series. `https://www.maxeler.com/products/mpc-nseries/`.

[30] memaslap - Load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memaslap.html`.

[31] Mininet. `http://mininet.org/`.

[32] ModelSim. `https://www.mentor.com/products/fv/modelsim/`.

[33] Jad Naous, David Erickson, Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Symposium on Networked Systems Design and Implementation*, pages 1–9. ACM, 2008.

[34] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. SDA: Software-defined accelerator for large-scale DNN systems. In *Hot Chips Symposium*, pages 1–23. IEEE, Aug 2014.

[35] Gregor N Purdy. *Linux iptables Pocket Reference*. O'Reilly Media, 2004.

[36] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Int. Symp. on Computer Architecture*, pages 13–24. IEEE, Jun 2014.

[37] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*. USENIX Association, 2012.

[38] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *Symposium on Operating Systems Principles*. ACM, 2013.

[39] SDNet. `https://www.xilinx.com/products/design-tools/software-zone/sdnet.html`.

[40] Amazon Web Services. EC2 Instances (F1) with Programmable Hardware. `https://goo.gl/fmEQPK`.

[41] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.

[42] Vivado Simulator. `https://www.xilinx.com/products/design-tools/vivado/simulator.html`.

[43] Satnam Singh and David J. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines*, pages 3–12. IEEE, April 2008.

[44] Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Noa Zilberman, Richard Clegg, Luo Mai, Richard Mortier, Peter Pietzuch, Jon Crowcroft, and Andrew W. Moore. Extending programs with debug-related features, with application to hardware development. *CoRR*, abs/1705.09902, 2017. URL: `http://arxiv.org/abs/1705.09902`.

[45] NetFPGA SUME Reference Switch. `https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-Learning-Switch`.

[46] Yuta Tokusashi and Hiroki Matsutani. A Multilevel NOSQL Cache Design Combining In-NIC and In-Kernel Caches. In *Symposium on High-Performance Interconnects*, pages 60–67. IEEE, Aug 2016.

[47] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA : A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, pages 122–135. ACM, April 2017.

[48] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex Event Detection at Wire Speed with FPGAs. *VLDB Endow.*, 3(1-2):660–669, September 2010.

[49] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, Sept 2014.

[50] Noa Zilberman, Matthew P Grosvenor, Diana Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. Where has my time gone? In *Passive and Active Measurement*, pages 201–214. Springer, March 2017.

[51] Noa Zilberman, Philip M. Watts, Charalampos Rotsos, and Andrew W. Moore. Reconfigurable Network Systems and Software-Defined Networking. *Proceedings of the IEEE*, 103(7):1102–1124, July 2015.

[52] Data Plane Development Kit. `http://dpdk.org/`.