An Open-Source Approach to Bridging the Gap in Network Hardware and Software

P.SHALLINI[1],V.LAKSHMI[2],LALLAM VASU[3],
ASSISTANT PROFESSOR[1,2,3],
DEPARTMENT OF ECE
PBR VISVODAYA INSTITUTE OF TECHNOLOGY AND SCIENCE::KAVALI

Abstract—

As network speeds increase to the tens of Gigabits per second range, it will become more difficult to design packet processing software capable of handling such massive data volumes. As a result, it is clear that there is a need for an appropriate open-source system that may serve as a prototype platform for testing new network capabilities while guaranteeing line-rate processing, precise timestamping, or decreased power usage. All of these needs may be met using hardware-based systems like NetFPGA, rather than only software. The primary barrier to adopting such an open-source FPGA-based solution is the time and effort needed for its creation. With the proliferation of HLL-based circuit synthesis tools, it is now possible to create hardware-based networking apps with a manageable learning curve, in comparison to the usage of HDLs in the past. In this article, we discuss how the new programming paradigm of FPGAs made possible by state-of-the-art High-Level Synthesis tools can feed current open-source hardware-based platforms for networking applications. We contrasted the time and effort required to construct a network flow monitor using conventional hardware development methods with the time and effort required to develop the same thing using High-Level Languages. Initial findings are quite encouraging, especially since the development period has been cut from months to weeks.

Key words

Network Functions on Field-Programmable Gate Arrays, High-Level Language, Hardware Description Language, Packet Processing, High-Speed Networks, High-Level Synthesis, Network Flow Monitor.

TRANSITION TO D ATA LINK

The capacity of communication networks is expanding rapidly as a result of technological advancements. Current installations use interfaces of 10 Gbps, although 40 and even 100 Gbps are becoming more common. Packet processing programmes need to be deployed within such high-speed networks in order to perform several network activities. Security (including firewalls, IDS/IPS, and legal interception) and network performance are two such examples (to analyse delay, jitter, loss, or throughput). The processing infrastructure must be adaptable enough to accommodate application updates in a timely manner. For the time being, it is most practical to employ software that runs on conventional x86 processors due to the large pool of readily accessible software engineers, the simplicity of the method, the short development cycles, and the inherent adaptability of software. Newer network bit rates place strict demands on performance, which are now unmet by software-only solutions. High performance network drivers are the foundation of open-source software for super-fast networks (e.g. Packets Hader, PFRing, or Intel DPDK). They perform well at 10 Gbps on today's top-of-the-line commodity gear. It is challenging to attain throughputs exceeding 10 Gbps reliably without packet losses, since access speed between applications and network devices is presently constrained. High and unpredictable latency may result from the several hops that each packet must make, rendering it unfit for use in applications like high-frequency trading. As a final point, software driver-based timestamping introduces inaccuracy and jitter since it is performed in batches rather than individually on individual packets. As a result, these drivers cannot guarantee line-rate low-latency processing at higher speeds or provide precise packet timestamping when required [1]. The offloading of some or all of the packet-processing application to the network device is an option when the performance of software-based solutions falls short of expectations. Looking back at the evolution of networking technology, it's clear to see that specialised hardware has always been used in state-of-the-art packet processing gadgets. In reality, many NICs already offload protocol operations (such TCP and IPSec) to improve system efficiency by taking over duties normally performed by software. Unfortunately, there is very little room for customization in these systems due to their restricted nature. With NetFPGA [2], you may create high-performance, open-source hardware while offloading less important duties to software on an x86 CPU.

WORK ON THE NETFPGA-10G

The open-source NetFPGA project highlights the growing popularity of FPGA-based systems for networking packet processing applications. NetFPGA was originally developed as a research and teaching tool, but it has now found widespread use in the academic community as a means of quickly prototyping novel approaches to future network architecture. Stanford University and Xilinx Research Labs [2] created NetFPGA with contributions from the community. The core components of NetFPGA-10G [5], the second-generation edition, are a Xilinx Virtex-5 FPGA, four full-duplex 10 Gbps Ethernet connections, and a PCI Express card. The platform has two different memory banks in addition to the FPGA's internal memory (Block RAMs, with 18 Kbits per block)

for supporting a wide range of network applications. In contrast to the second type's 288 MB of low-latency dynamic RAM, which is designed for packet buffering, the first type's 27 MB of high-speed static RAM is optimised for rapid lookup tables. The board can talk to a host computer using PCI Express Gen 1 lanes. However, the NetFPGA 10G platform just requires a 12 V power supply, thus it may operate independently (i.e., without being attached to any PCI Express socket); this may be the best option for running line-rate applications with a very low power consumption. An overview of the hardware's layout is shown in Fig. 1. As an open-source platform, the NetFPGA-10G is useful for researchers who want to create network applications that make use of FPGAs. The developer community has access to a central database where they may store and trade code, binaries, and other resources used in the creation of software. The Xilinx Embedded Development Kit is used in the creation of NetFPGA-10G applications (EDK). Projects using the EDK are split into two categories: a) those that focus on the hardware platform implemented on the FPGA, and b) those that focus on the software performed by the embedded processor on the FPGA. The hardware foundation relies on components such as Ethernet MAC, PCI Express interface (PCIe), integrated soft processor (MicroBlaze), direct memory access (DMA), and user-created modules. The designer decides which hardware cores should be included in the FPGA platform. The embedded processor is responsible for running the EDK project's software and plays a crucial role on the NetFPGA-10G since it is responsible for configuring the Ethernet ports. While an EDK project is used to build hardware and software for the FPGA, a NetFPGA-10G project also contains software code (such as drivers and user applications) that will run on the FPGA.

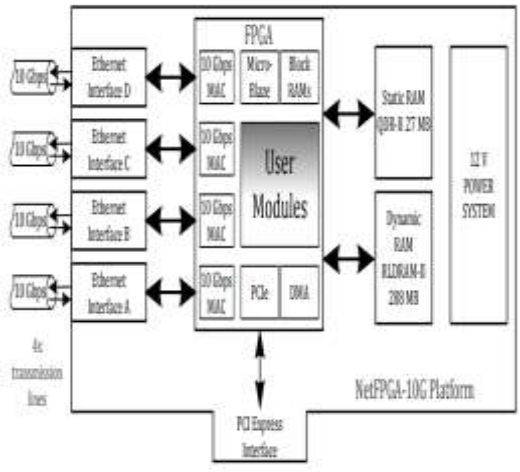

Fig. 1: NetFPGA-10G structure.

produced by an x86 CPU. For connectivity between the FPGA and the x86 host computer, PCI Express is used. Together, these parts provide the infrastructure required to create open-source network applications that take full use of the power of modern technology. Downloading the most recent iterations of the available projects from the public repository is the first step in a typical design cycle for network applications on the NetFPGA-10G, as illustrated in Fig. 2. (task 1). As a starting point for the new design, the developers must choose one that works best for them. The goal here is to find ways to recycle existing features so that more time can be spent actually building the new thing. It is necessary to choose the host computer, if any, and the FPGA software to execute first. To make such a decision is to strike a balance between the development time and the certainty in the performance (number of clockscyles it takes to execute) of each job. Following an HDL design flow that includes Verilog or VHDL codification and validation, developers will next construct their own hardware modules (task 2.a) if they are not already accessible in the repository; we emphasise that this is the most time-consuming stage in the design flow. To manage every Ethernet interface, as many of these hardware modules as are required may be made. When all of the modules have been developed or modified, the next step is integration (task 2.b), which involves linking them together using on-chip communication protocols. The last stage of designing an FPGA embedded system is, if necessary, changing the embedded processor's executable programme (task 2.c). After the hardware and embedded software are ready to execute on the FPGA, the following step of design is the creation of the host computer operations that are not time essential (task 3). A Linux PCI Express driver is downloadable from the NetFPGA-10G repository, and may be used as-is or modified to suit your needs. In addition, user-level programmes written in the traditional C/C++ software development cycle may work in tandem with the aforementioned driver to tackle relatively low-speed tasks. Developers may release their work to the community whenever they are satisfied with the design's functioning and have tested it extensively (task 4).

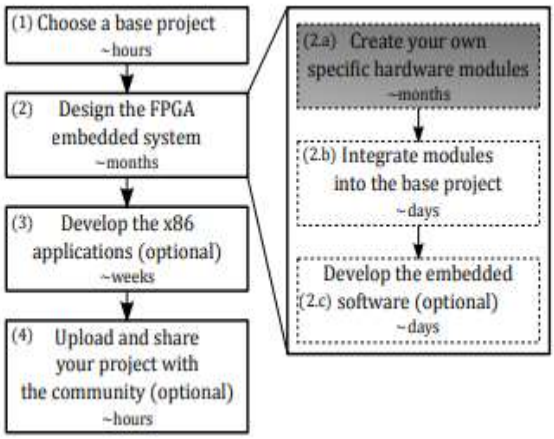

Fig. 2: Typical design flow in the NetFPGA platform.

What follows is a breakdown of the time required for development: Task 2.a requires many months (70%-90% of the entire development time) and is thus the most time-consuming. Second, there's Task 2.c, which may take a few days to complete (depending on the applica tion), and then there's Task 3, which, if implemented, would take many weeks. Most of the remaining work may be completed in a few hours by an experienced engineer. Finally, the expense of software-based advancements is substantially greater in terms of person months. The FPGA programming methodology makes the development of custom hardware modules (task 2.a) the most time-consuming part of the process. The Register Transfer Level (RTL) is the highest level of HDL, and it is used to build circuits using a model that includes information on the flow of data and time (RTL). Because HDLs give a higher degree of abstraction than the circuit that ultimately executes on the FPGA, the designer benefits from this fact. HDL synthesis tools convert RTL model transfer functions between registers into logic gates, but the hardware registers maintain a one-to-one correlation with their HDL RTL model counterparts. Therefore, the time required to establish an HDL design is much longer than that required for software solutions, since HDL codification entails fixing a-priori the architecture of the hardware being implemented. As a result, FPGAs have not found widespread use in the field of networking. Time spent on job 2.a must be shortened if we are to close the gap between software and hardware network advancements and enjoy the best of both worlds.

SAVING THE DAY: HIGH-LEVEL LANGUAGES

Reduce hardware development time using modern High-Level Synthesis (HLS). High-Level Synthesis (HLS) tools modify the FPGAs' programming paradigm, allowing for the incorporation of HLL at the design capture stage. As a result, they water down the distinction between a CPU and an FPGA's programming model [6]. Different kinds of High-Level Languages exist, from graphical descriptions to ad hoc languages built from extensions of more conventional ones. Although HLS as a concept has been developed over many years [7], only in the last few years have new promising and effective tools become available. Quick progress is being made in the electrical sector towards the widespread use of these HLS-based technologies. Many of them can take an ANSI-C, C++, or SystemC source file and generate HDL code. There are a number of reasons why C/C++ have been so successful as a design entry. For starters, there is a lot of pre-existing code and almost all computer and electrical experts are already acquainted with them. In many application domains, including networking, C/C++ is the language of choice for prototyping and development. In addition, it is a logical method for Hardware/Software co-design, beginning with a software programme and then migrating to the hardware those components that need additional performance, while still making use of the same language. Ca dance's C-to-Silicon, Synopsys's Symphony C Compiler, Calypte's Catapult C, Impulse's Codeveloper, Xilinx's Viv ado-HLS, Blue spec's BSC (Blue spec Compiler), Jacquard computing's ROCCC 2.0 (Riverside Optimizing Compiler for Config durable Computing), and blue spec's.

How HLL may aid in the hardware design process

When using HLL, the first stages of implementation are completed significantly more quickly, and more of the possible design space may be investigated in a shorter amount of time. Figure 3 suggests that software simulation is rapid enough to go to the next iteration of the design process. Therefore, there is no need for the time-consuming and complicated HDL simulation stage.

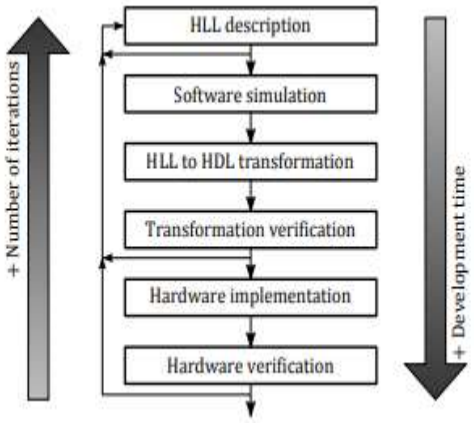

Fig. 3: Hardware design flow using High-Level Langauges.

The hardware description language (HDL) conversion from high-level language (HLL) to hardware description language (C-to-hardware) is also very effective and gives more insight into the hardware's performance (number of cycles spent in the execution, maximum frequency, area usage, etc.). As a result, the designer may experiment with several design possibilities or implement additional features—all of which are more expensive when implemented using HDLs—and get rapid feedback. One of the most common complaints levelled towards HLS tools is that, although they do cut down on design time, they also compromise performance by preventing architects from fine-tuning the design at the lowest levels. Given the expressiveness of HLL and the shorter development time, these statements are debatable, as shown by [6], [8]. As a result, the designers have access to a significantly bigger design area than they would have using the HDL method. While hardware description languages (HDLs) may also provide similar improvements, the programming paradigm that revolves on RTL description requires the implementation of a static architecture and hence precludes future optimizations without rewriting the code. In addition, HLLs obfuscate all implementation details that aren't crucial to performance (such as optimising state machines, timing closure issues, resource allocation and scheduling, etc.), freeing the designer to concentrate on system-level performance problems (such as processes communicating with one another or storing data) that have a greater impact on performance as a whole.

The HLL development methodology for building hardware for use in networking software

Unfortunately, the hardware characteristics and parallelism required to create NetFPGA applications are not built into the C/C++ programming language. Having these more elements adds more complexity. Unfortunately, there is not yet much uniformity in the HLS tools' approaches to these problems. As of right now, a well-executed hardware design is not the result of a generic C/C++ code but rather of a code that has been adapted for a certain architecture.

We utilised Xilinx's Viv ado-HLS software for this project [9]. This programme can take an algorithm model written in normal C/C++ and generate an HDL description suitable for usage in Xilinx FPGAs (such as the one present on the NetFPGA-10G). In addition, the Viv ado HLS tool creates HDL-code-free hardware cores that may be dropped into an EDK project. This tool can create circuits that are time-accurate since it takes into account both the clock frequency and the intended device. The delay associated with each job, as measured in terms of clock cycles, is reported. Consequently, much like HDL-designed hardware, there is no jitter in the tasks themselves (for example, when timestamping packets). So-called directives (#pragma statements) may be used to take use of parallelism, pipelines, regulate latency, specify interfaces, and other hardware characteristics if the processing requirements are not met by the original code. Even though the tool creates a module for each clock domain, dual-clock FIFOs may be used at the EDK level to glue the created cores for each domain together when several clock domains are present. The usage of HLL simplifies the most complicated and time-consuming phase of developing the processing logic executed on each packet. For instance, a dual-clock FIFO may be used to interface the 10G-MAC clock domain to the DMA domain if an application needed to analyse all the Ethernet packets it received and deliver aggregated information to a software layer on the x86 machine. HLL model capture will be used for the development and verification of all user-added intelligence (i.e., processing and communication).

CONCLUSION

Compared to solutions based on commodity x86 hardware, the performance and predictability of packet processing systems developed in FPGA are clearly superior. However, most network engineers find it unappealing due to the time and money needed for development. Surprisingly, the advent of new High-Level Synthesis tools offers hope for overcoming these challenges. Current FPGA-based platforms, such as the free and open-source NetFPGA-10G, may benefit greatly from the use of state-of-the-art HLS tools, as we have shown here. In addition, we have described the most significant obstacles that prevent High-Level Languages from gaining general acceptance. FPGAs now have a programming paradigm that makes it possible to capture designs using HLL, cutting down application development time from months to weeks when compared to a conventional hardware development flow based on hardware description languages (HDLs).

This article demonstrates how to create hardware-based network applications without familiarity with HDLs via the production of flow records at 10 Gbps line-rate. In addition, the performance and hardware resource utilisation of solutions developed using a high-level design process were found to be satisfactory. In doing so, the groundwork is laid for an HLL-based (usually C/C++) application framework for packet processing. The framework would further abstract hardware specifics, enabling the traditionally wide gap between software and hardware development in networking applications to be closed.

REFERENCES

[1] V. Moreno, P. Santiago del Rio, J. Ramos, J. Garnica, and J. GarciaDorado, "Batch to the Future: Analysing Timestamp Accuracy of High-Performance Packet I/O Engines," Communications Letters, IEEE, vol. 16, no. 11, pp. 1888 – 1891, november 2012.

[2] M. Blott, J. Ellithorpe, N. McKeown, K. Vissers, and H. Zeng, "FPGA Research Design Platform Fuels Network Advances," Xcell Journal, pp. 24–29, 2012.

[3] J.-P. Deschamps, G. Sutter, and E. Cant, Guide to FPGA Implementation of Arithmetic Functions, ser. Lecture Notes in Electrical Engineering. Springer, 2012, vol. 149. [Online]. Available: http://dx.doi.org/10.1007/978-94-007-2987-2

[4] J. Schonw ¨ alder, A. Pras, and J.-P. Martin-Flatin, "On the future ¨ of Internet management technologies," Communications Magazine, IEEE, vol. 41, pp. 90–97, Oct 2003.

[5] "NetFPGA-10G board description," 2012. [Online]. Available: http://netfpga.org/10G specs.html

[6] Xilinx Inc., Introduction to FPGA Design with Vivado HighLevel Synthesis. UG998, July 2013. [Online]. Available: http://www.xilinx.com/support/

[7] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," IEEE Design & Test of Computers, vol. 26, no. 4, pp. 18–25, 2009.

[8] A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in Reconfigurable Computing: Architectures, Tools and Applications. Springer, 2011, pp. 67–78.

[9] Xilinx Inc., Vivado Design Suite User Guide. HighLevel Synthesis. UG902, July 2012. [Online]. Available: http://www.xilinx.com/support/

[10] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in ACM SIGCOMM Computer Communication Review, vol. 34, no. 4. ACM, 2004, pp. 245–256.

[11] M. Forconesi, G. Sutter, and S. Lopez-Buedo, "Open source code of nf bram and nf qdr," 2013. [Online]. Available: https://github.com/hpcn-uam/HW-Flow-Based-Monitoring

[12] M. Forces, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and flexible flow-based monitoring for high-speed networks," Field Programmable Logic and Applications, 2013.