Juni Khyat
(UGC Care Group I Listed Journal)

ISSN: 2278-4632
Vol-12 Issue-01 2022

Tools for Interfacing Hardware and Software in Open Source Networks

KURELLA AYYAPPA RAVI KIRAN[1], Mr. MOLUGUMATI PREMCHAND[2],Mrs. G. NAVYA[3]
ASSISTANT PROFESSOR[1,2,3]
DEPARTMENT OF ECE, SWARNANDHRA COLLEGE OF ENGINEERING AND TECHNOLOGY, NARASAPUR

Abstract—

It will be more challenging to develop packet processing software that can manage data quantities in the tens of gigabits per second range as network speeds continue to rise. Consequently, it's evident that a suitable open-source solution is required to test new network capabilities on a prototype platform, ensuring either reduced power consumption, accurate timestamping, or line-rate processing. It is possible to address all of these requirements using hardware-based solutions, such as NetFPGA, instead of relying only on software. The time and effort required to create such an open-source FPGA-based solution is the main obstacle to its adoption. Since HLL-based circuit synthesis tools have proliferated, it is now feasible to build hardware-based networking programs with a more manageable learning curve than was previously the case with HDLs. As a result of cutting-edge High-Level Synthesis tools, field-programmable gate arrays (FPGAs) have emerged as a new paradigm in computer programming. This article explores how this paradigm might augment existing open-source hardware-based platforms used for networking applications. In this study, we compared the development times of a network flow monitor built using traditional hardware development techniques with those built using High-Level Languages. The first results are promising, particularly considering the short development time (only weeks instead of months).

Key words

Functional Networks on Field-Programmable Gate Arrays, High-Level Synthesis, Packet Processing, High-Speed Networks, Hardware Description Language.

Swap to D-AT-Link

New technologies are quickly increasing the capacity of existing communication networks. Despite the increasing use of 40 Gbps and even 100 Gbps, 10 Gbps ports are currently used in deployments. In order to carry out various network operations, packet processing programs must be implemented in such high-speed networks. Two examples are network performance (to analyze latency, jitter, loss, or throughput) and security (including firewalls, IDS/IPS, and lawful interception). Application changes must be accommodated in a timely way by the processing infrastructure. Because of the abundance of available software engineers, the ease of the approach, the rapid development cycles, and the inherent flexibility of software, it is now most practicable to use software that runs on traditional x86 processors. Performance requirements imposed by newer network bit rates are so high that software-only solutions can no longer keep up. Packets Hader, PFRing, and Intel DPDK are just a few examples of the open-source software for very fast networks that rely on high performance network drivers. At 10 Gbps, they work admirably on the most cutting-edge commodity hardware available today. Access speeds between applications and network devices are still limited, making it difficult to securely achieve throughputs above 10 Gbps without packet losses. Due to the several hops that each packet must travel, the latency may be significant and unpredictable, making it unsuitable for applications such as high-frequency trading. Finally, since it is executed in batches instead than on individual packets, software driver-based timestamping adds jitter and inaccuracy. Consequently, these drivers aren't up to the task of providing accurate packet timestamps when needed or ensuring line-rate low-latency processing at greater rates [1]. As a fallback strategy when software-based solutions fail to meet expectations, the packet-processing application might be partially or fully offloaded to the network device. When we examine the history of networking technology, we can see that cutting-edge packet processing devices have always made use of specialized hardware. Actually, a lot of network interface cards (NICs) already do things like offload protocol operations (like TCP and IPSec) to make the system more efficient by doing things that software ordinarily does. Due to their limited nature, these systems unfortunately do not allow for much flexibility. By using NetFPGA [2], you may build open-source hardware with great performance while letting software run on an x86 CPU do less critical tasks.

NETFPGA-10G Project

One example of the rising need for FPGA-based solutions is the open-source NetFPGA project, which processes packets in networks. The academic community has discovered extensive use of NetFPGA, which was first created as a tool for research and education, as a way to rapidly prototype new ideas for future network design. With help from the community, NetFPGA was established by Xilinx Research Labs and Stanford University [2]. The second-generation version of NetFPGA, NetFPGA-10G [5] relies on a PCI Express card, four full-duplex 10 Gbps Ethernet connections, and a Xilinx Virtex-5 FPGA. The platform's internal memory (Block RAMs, with 18 Kbits per block) and two external memory banks (DDR2 and SDRAM) allow it to serve a diverse set of network applications. The first kind has 27 MB of high-speed static RAM that is optimized for speedy lookup tables, while the second type has 288 MB of low-latency dynamic RAM that is built for packet buffering. When connected to a host computer, the board can communicate over PCI Express Gen 1 lanes. Alternatively, for very low-power line-rate applications, the NetFPGA 10G platform may be the ideal choice as it operates autonomously (i.e., without being connected to any PCI Express socket) and only needs a 12 V power source. Figure 1 shows a schematic representation of the hardware. Researchers interested in developing FPGA-based network applications may find the open-source NetFPGA-10G platform beneficial. Code, binaries, and other resources used in software development may be stored and traded in a central database that is accessible to the developer community. Making apps for the NetFPGA-10G requires the Xilinx Embedded Development Kit. There are two types of EDK projects: those that concentrate on the FPGA's hardware platform and those that center on the embedded processor's software. Direct memory access (DMA), user-created modules, Ethernet MAC, PCI Express interface (PCIe), and integrated soft processor (MicroBlaze) are part of the hardware base. When developing an FPGA platform, the designer chooses which hardware cores to include. On the NetFPGA-10G, the embedded processor is vital for setting the Ethernet ports and executing the software for the EDK project. Software code (such as drivers and user applications) that will operate on the FPGA is also included in a NetFPGA-10G project, in contrast to an EDK project.

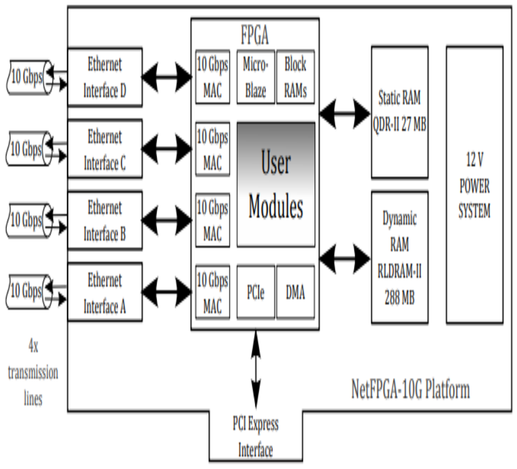
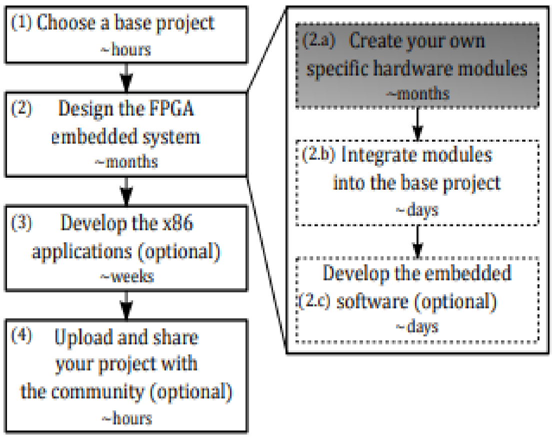


Fig. 1: NetFPGA-10G structure.

created by a central processing unit (CPU) using an x86 architecture. A PCI Express cable connects the FPGA to the x86 host system. All of these pieces work together to provide the framework for making open-source network apps that harness the potential of today's technologies. As shown in Figure 2, the first stage of a standard design cycle for network applications on the NetFPGA-10G is to download the latest versions of the accessible projects from the public repository. duty one. The developers should choose a layout that suits them best to use as a foundation for the new design. To free up more time to construct the new object, we need to figure out how to reuse parts of the old one. Prior to running any FPGA program, you must choose the host computer (if applicable). Finding a happy medium between development time and performance certainty (number of clock cycles required to complete each task) is the key to making this kind of choice. We stress that the most time-consuming part of the design flow occurs after an HDL design flow that incorporates Verilog or VHDL codification and validation; developers will build their own hardware modules (task 2.a) if they aren't already available in the repository. It is possible to manufacture an unlimited number of these hardware modules in order to handle all Ethernet interfaces. Integration (task 2.b) entails connecting all of the modules that have been produced or changed so far by means of on-chip communication protocols. As a final step in developing an FPGA embedded system, job 2.c involves making any required changes to the executable program of the embedded processor. Step three of the design process involves making non-time-critical host computer activities once the embedded software and hardware are prepared to run on the FPGA. If you want to utilize an existing Linux PCI Express driver or make some changes to it, you can find it in the NetFPGA-10G repository. Furthermore, the aforementioned driver may be used in conjunction with user-level programs created via the conventional C/C++ software development cycle to handle somewhat slow-paced activities. Once developers

ISSN: 2278-4632
Vol-12 Issue-01 2022

are happy with the design's functionality and have tested it thoroughly, they may release it to the community


(1) Choose a base project
~hours
(2) Design the FPGA embedded system
~months
(3) Develop the x86 applications (optional)
~weeks
(4) Upload and share your project with the community (optional)
~hours
(2.a) Create your own specific hardware modules
~months
(2.b) Integrate modules into the base project
~days
(2.c) Develop the embedded software (optional)
~days

(task 4).

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

The time needed for development is broken down as follows: The most time-consuming part of the development process is Task 2.a, which takes a long time (around 70% to 90% of the total time). The second one is Task 2.c, which may be finished in a few days (depending on the application), and the third one would take weeks to implement if done. An expert engineer could do most of the remaining tasks in a few of hours. Lastly, improvements in software are far more expensive in terms of man-months. Custom hardware module development (task 2.a) is the most time-consuming portion of the procedure when using the FPGA programming technique. The pinnacle of HDL is the Register Transfer Level (RTL), which is used for circuit construction using a model that incorporates data and time flow information (RTL). The designer gains an advantage from using HDLs since they provide a greater level of abstraction compared to the circuit that runs on the FPGA in the end. Although hardware registers maintain a one-to-one connection with their HDL RTL model equivalents, logic gates are created from RTL model transfer functions between registers using HDL synthesis tools. The time needed to build an HDL design is much greater compared to software solutions since HDL codification involves setting the hardware architecture in advance. Consequently, FPGAs are still underutilized in the networking industry. Getting the most out of both the software and hardware aspects of network developments requires cutting down on time spent on task 2.a.

Helping Hands: Advanced Languages

Utilize state-of-the-art High-Level Synthesis (HLS) to lengthen the duration of hardware development. In order to include HLL at the design capture stage, High-Level Synthesis (HLS) tools alter the programming paradigm of FPGAs. Consequently, they make it harder to tell an FPGA's programming model apart from a CPU's [6]. From graphical representations to ad hoc languages constructed from extensions of more traditional ones, there are several forms of high-level languages. The idea of HLS has been around for a while [7], but new, promising tools have just become accessible in the last several years. The electrical industry is racing forward to implement these HLS-based solutions widely. A large number of them are capable of generating HDL code from source files written in ANSI-C, C++, or SystemC. C/C++ has been so successful as a design entry for many reasons. To begin, there is a great deal of existing code, and almost all electrical and computer specialists are already familiar with it. C++ and C are the languages of choice for development and prototyping in many application fields, including networking. Additionally, it is a reasonable approach to Hardware/Software co-design, starting with a software program and then moving to hardware those components that need more performance, all while maintaining the same language. The following compilers are available: C-to-Silicon from Ca dance, Symphony C Compiler from Synopsys, Catapult C from Calypte, Codeveloper from Impulse, Viv ado-HLS from Xilinx, BSC from Blue Spec, ROCCC 2.0 from Jacquard Computing, and more.

The potential benefits of HLL for hardware design

Using HLL, the first phases of implementation are finished more faster, and a larger portion of the design space may be explored in less time. Software simulation seems to be fast enough to go on to the next design iteration,

ISSN: 2278-4632
Vol-12 Issue-01 2022

as seen in Figure 3. As a result, the laborious and complex HDL simulation step is unnecessary.

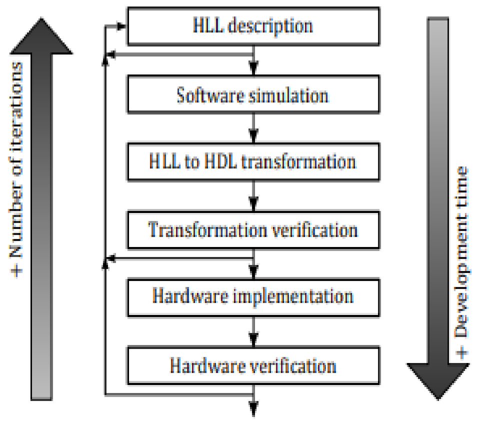


Fig. 3: Hardware design flow using High-Level Langauges.

More information about the hardware's performance (execution cycles, maximum frequency, area utilization, etc.) may be obtained using the very effective C-to-hardware (high-level language to hardware description language) translation. Consequently, the designer may integrate more features or try out many design options—all of which are costly when done using HDLs—and get quick feedback. Although HLS tools do save design time, a typical criticism is that they hinder performance by limiting architects' ability to fine-tune the design at the lowest levels, which is a major concern. These claims are controversial because of the expressiveness of HLL and the reduced development time (as shown by [6], [8]). Consequently, the designers may work with a much larger area for design than they could using the HDL approach. The RTL-dedicated programming paradigm necessitates a static architecture, which in turn prohibits future optimizations from occurring without rewriting the code, even if HDLs may provide comparable benefits. Furthermore, HLLs hide all implementation details that aren't performance-critical, allowing the designer to focus on system-level performance issues, like processes communicating or storing data, rather than optimizing state machines, timing closure issues, resource allocation and scheduling, etc.

Methodology for creating hardware for use in networking software that is based on the HLL

The C/C++ programming language lacks the built-in hardware features and parallelism needed to develop NetFPGA applications. Adding these additional components makes things more complicated. The current state of the HLS tools' responses to these difficulties is disappointingly lacking in standardization. At the moment, optimized C/C++ code is more likely to produce a poorly implemented hardware design than generic code that has been tailored to a certain architecture.

The Viv ado-HLS software from Xilinx was used in this project [9]. An algorithm model written in standard C/C++ may be converted to an HDL description used in Xilinx FPGAs (such the NetFPGA-10G) by means of this application. Also, you may put HDL code-free hardware cores made using the Viv ado HLS tool into an EDK project. Because it considers the clock frequency in addition to the target device, this tool may generate time-accurate circuits. Each task's corresponding delay, expressed in terms of clock cycles, is detailed. There is no jitter in the jobs themselves, for example while timestamping packets, much as in HDL-designed hardware. If the original code cannot handle the processing needs, so-called directives (#pragma statements) may be used to take use of pipelines, manage latency, define interfaces, and other hardware features. When dealing with multiple clock domains, it is possible to employ dual-clock FIFOs at the EDK level to combine the cores that are formed for each domain, even if the tool builds a module for each clock domain. Developing the processing logic that is done on each packet is a complex and time-consuming operation, but using HLL simplifies it. If an application received Ethernet packets and wanted to aggregate them, it could use a dual-clock FIFO to connect the 10G-MAC clock domain to the DMA domain and then send the aggregated data to a software layer on the x86 processor. When creating and testing any intelligence that the user adds, such as processing or communication, HLL model capture will be used.

CONCLUSION

Performance and predictability of FPGA-developed packet processing systems are noticeably better than those of solutions built on commodity x86 hardware. Nevertheless, the development costs and time commitment make it unsuitable to the majority of network engineers. Unexpectedly, new High-Level Synthesis tools provide optimism for overcoming these obstacles. We have shown that cutting-edge HLS tools may be very useful for current FPGA-based systems, such the open-source and free NetFPGA-10G. We have also detailed the primary challenges that people face while trying to adopt High-Level Languages. A new programming paradigm for field-programmable gate arrays (FPGAs) allows for the capturing of designs using hardware description languages (HDLs), which reduces the application development time from months to weeks compared to the traditional hardware development flow based on HDLs.

Using the creation of flow records at 10 Gbps line-rate, this article shows how to build hardware-based network applications without knowing anything about HDLs. Results from solutions built using a high-level design methodology were also good in terms of performance and hardware resource utilization. Then, a framework for packet processing applications built on top of HLL, typically in C/C++, may be put in place. By further abstracting hardware characteristics, the framework would make it possible to bridge the gap between software and hardware development in networking applications, which has historically been rather large.

REFERENCES

In their November 2012 article "Batch to the Future: Analysing Timestamp Accuracy of High-Performance Packet I/O Engines," published in the IEEE Communications Letters, V. Moreno, P. Santiago del Rio, J. Ramos, J. Garnica, and J. GarciaDorado discuss batch processing and the accuracy of timestamps.

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

[3] Guide to FPGA Implementation of Arithmetic Functions, edited by J.-P. Deschamps, G. Sutter, and E. Cant, published in the Lecture Notes in Electrical Engineering series. Chapter 149, published by Springer in 2012. This publication is accessible online at: http://dx.doi.org/10.1007/978-94-007-2987-2.

In October 2003, J. Schonwälder, A. Pras, and J.-P. Martin-Flatin published an article titled "On the future ¨ of Internet management technologies" in the IEEE Communications Magazine, volume 41, pages 90-97.

The "NetFPGA-10G board description" from 2012 is on page 5. The information may be found online at: http://netfpga.org/10G specs.html.

[6] Xilinx Inc., UG998, July 2013, Introduction to FPGA Design using Vivado High-Level Synthesis. Accessible online at: http://www.xilinx.com/support/.

(7) "High-level synthesis: Past, present, and future," in 2009's IEEE Design & Test of Computers, vol. 26, no. 4, pp. 18-25, written by G. Martin and G. Smith.

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

[9] Instructions for Using the Vivado Design Suite by Xilinx Inc. 2012 July, High-Level Synthesis (UG902). The Internet. Please visit: http://www.xilinx.com/support/ for further information.

"Building a better NetFlow" (in ACM SIGCOMM Computer Communication Review, vol. 34, no. 4, ACM, 2004, pp. 245-256), written by C. Estan, K. Keys, D. Moore, and G. Varghese.

[11] "Open source code of nfbram and nfqdr," 2013 by M. Forconesi, G. Sutter, and S. Lopez-Buedo. The Internet. Here is the link: https://github.com/hpcn-uam/HW-Flow-Based-Monitoring.

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