Surveying P4 Compiler Development After 2016

Yue Wu, Henning Stubbe*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: yue02.wu@tum.de, stubbe@net.in.tum.de

***Abstract*—Software-defined networking (SDN) initially appeared as a new network management technology aimed at improving network performance. Since 2013, SDN associated with OpenFlow protocol (a communication protocol) has become an industry standard. However, SDN-related protocols need to specify their headers on the hardware device they operate, which limits the flexibility for targeting different devices and increases the complexity for future protocol expansion. To address this problem, the P4 language was introduced as a protocol-independent programming language for describing the process of network data packets and now has been widely used in many different devices, such as Application-specific integrated circuit (ASIC), Field-programmable gate array (FPGA), Network interface card (NIC), CPU etc. through the corresponding compiler. The purpose of this survey is to present the latest varieties of P4 compilers, including their respective characteristics and target equipment.**

***Index Terms*—P4 compiler, P4FPGA, P4LLVM, T4P4S, p4c-XDP**

1. Introduction

Nowadays, as the requirements for network performance increase, more network equipment is needed which leads to more and more cumbersome configuration of traditional equipment [1]. To prevent this trend, next generation networks are supposed to have the following characteristics: programmable customization on demand, centralized and unified management, dynamic traffic supervision and automated deployment [2]. That is why the concept of SDN was born. SDN is physically separated into a control plane and a forwarding plane [3]. The former provides the intelligent logic in network equipment, which controls how to manage data traffic, while the latter manages forwarding/manipulating/discarding network data traffic. This improvement will lead to a more efficient configuration process when modifying different network behavior, since the control plane is the only part to change.

However, the fact that only the control plane can be used for programming could still raise some problems. Under normal circumstances, data packets in the forwarding process are solidified by the forwarding chip of the device that usually does not support protocol expansion. In addition, the cost of developing new forwarding chips that support new protocols or extended protocol features is also very expensive. The need to design such hardware will lead to a series of problems, such as high update costs and long development time. Therefore, the new generation of SDN solutions should enable the forwarding plane to be programmable as well, so that the software can truly define the network and network equipment. P4 provides users with this function, which breaks the limitations of the hardware devices on the forwarding plane and allows programming to control the analysis and forwarding process of data packets. Therefore, the network and devices are "open" to users from top to bottom. Section 2 will provide a short introduction to the P4 language. After that, four different P4 compilers will be presented in Section 3. Section 4 will give an overview of the future research direction.

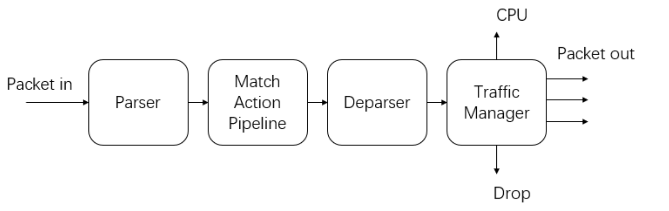

Figure 1: Example P4 Abstract Architecture [4]

2. P4 language

Before delving into the various P4 compilers, the P4 language will be introduced first.

The Programming Protocol-Independent Packet Processors (P4) is a Domain-Specific Language which was first proposed by Bosshart et al. in [3]. As a design goal, P4 is expected to achieve the following three design goals: 1) Protocol independence: Network equipment is not bound to any specific network protocol, and users can use P4 language to describe any network data plane protocol and packet processing behavior; 2) Target independence: Users do not need to care about the details of the underlying hardware to implement the programming description of the data packet processing method; 3) Reconfigurability: Users can change the program of packet parsing and processing at any time and configure the switch after compilation to truly implement on-site reconfiguration. In order to realize the above-mentioned goals, P4 language compilers are required to adopt a modular design, while the input and output of each module adopt standard configuration files. An abstract architecture of P4 is shown in Figure 1.

3. P4 Compilers

When P4 first appeared, it was still mainly oriented to the software control plane. In order to improve the performance of the programmable forwarding plane, a platform is needed which can help researchers to efficiently design on hardware. So far, hardware devices like ASIC, FPGA, NIC and CPU have been widely used, but been specified in their own programming language [5]. Therefore, P4 compilers show their importance in this case because they connect the P4 program with the underlying hardware that were initially unrelated. A typical P4 compiler has two main tasks: to generate the configuration at compile time to implement the data plane, and to generate the application programming interface (API) to populate tables [4].

3.1. General information and reference compiler

When the first version of P4 language $P4_{14}$ appeared, p4c-behavioral [6] was the standard P4 compiler, which used p4-hilr [7] to convert the source code to the P4 intermediate representation (IR). Typically IR is the data structure internally used by a compiler to represent source code. Later, $P4_{14}$ was found to have syntax and semantics problems [8]. In order to address these issues, a new version of P4 language $P4_{16}$ was released in 2016. Compared with the old version, in $P4_{16}$, a large number of language features have been transferred from the language to the libraries including counters, checksum units, meters, etc. As a result, the P4 language has been transformed into a more compact core language with libraries. p4c [9] is now the reference modular compiler for P4 that supports both $P4_{14}$ and $P4_{16}$. It can provide the target independent front-end compiler itself, and support different target specific backend compilers which will be introduced in the following subsections.

3.2. Target specific compiler - P4FPGA

FPGA is an ideal target platform for P4 due to its high degree of programmability [10]. However, the design of a compiler that converts P4 language into FPGA HDL code faces the following difficulties: 1) FPGA is mainly programmed through a low-level, non-portable code base; 2) Due to the differences between programs and different loading strategies adopted by different architectures, it is difficult to generate efficient hardware code implementation based on P4 source code; 3) Although the P4 language is not aware of the underlying hardware architecture, it relies on a series of “extern” syntax to import external valid functions, which makes code generation more complicated. To solve these problems, H. Wang et al. introduced the P4FPGA compiler in [4] which guarantees flexibility, efficiency and portability between the P4 program and FPGA device.

The proposed P4FPGA compiler reuses the reference P4 compiler p4c as its front-end to reduce engineering workload. As far as language version support is concerned, it can be used under $P4_{14}$ and $P4_{16}$ syntax, and can also be applied to different architecture configurations, which will be mentioned in 3.2.2. Figure 2 outlines the workflow of P4FPGA. Among them, code generation, P4FPGA runtime and optimization principles implemented as IR to IR transformers are the core parts.

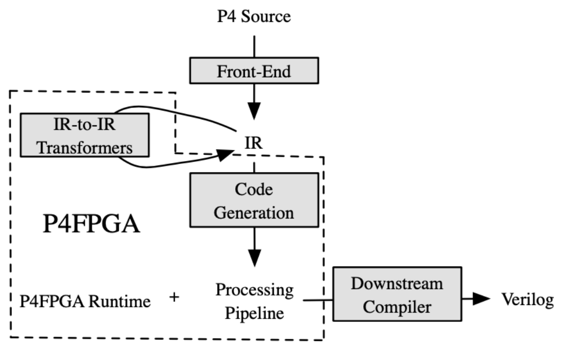

Figure 2: P4FPGA structure [4]

3.2.1. Code generation. In P4FPGA, the physical processing structure is generated as a block. These basic blocks are implemented in P4FPGA by using parameterized templates. During initialization, these templates are hardware modules used to implement the parser, match-action and deparser logic.

3.2.2. P4FPGA runtime. This is another important part of P4FPGA because it provides an efficient, flexible and scalable execution environment for the processing algorithms in P4. It defines a method that allows the generated code to access general-purpose functions through the untargeted abstraction, and provides an abstract architecture that can be implemented uniformly on different hardware platforms.

P4 programmers may write various network applications and put forward different requirements on the runtime. Therefore P4FPGA provides two architectures to support potential user scenarios: 1) Multi-port switching which is suitable for networking forward components like switches and routers and testing new network protocols; 2) Bump-in-the-wire which is suitable for network functions and network acceleration, but with only one input port and one output port.

3.2.3. Optimization principles. Finally, in order to ensure the high efficiency of generating code through P4FPGA, optimizations such as leveraging hardware parallelism in space and time to increase throughput, transforming sequential semantics to parallel semantics to reduce latency, using a resource-efficient component to implement match tables etc. are implemented in the context of the NetFPGA SUME platform in [4].

In summary, the P4FPGA includes a C++ based compiler along with a Bluespec-based [11] runtime system, as well as a p4c frontend and a custom backend. All source code is available at http://www.p4fpga.org.

3.3. Process optimizing compiler - P4LLVM

In addition to the perspective of a hardware-target backend compiler, there also exists compilers whose goals are optimization algorithms. P4 language with a better configuration framework can greatly shorten the packet processing time, thereby maximizing the use of network resources. P4LLVM was introduced by Dangeti et al.

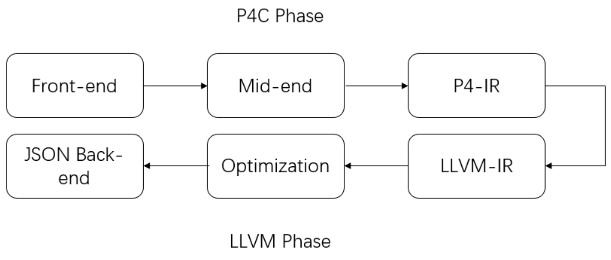

Figure 3: Flow of P4LLVM [12]

in [12] as a compiler based on the LLVM framework, and is proved to have a better performance than the standard p4c compiler.

LLVM is an abbreviation for Low Level Virtual Machine. It is a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs by providing high-level information to compiler transformations at compile-time, link-time, run-time and in the idle time between runs [13]. Just like p4c, the LLVM framework provides great convenience when plugging front-ends, back-ends and optimizations to different targets and accessing various machines. This advantage indicates that LLVM can be used in conjunction with P4, which is how P4LLVM invented.

P4LLVM only supports the $P4_{16}$ programs and reuses the p4c frontend module to check the lexical, syntactic and semantic correctness of the P4 code and mid-end module for preprocessing. After that, the intermediate representation of P4 (P4-IR) is converted to the intermediate representation of LLVM (LLVM-IR), then the IR is passed through various optimization sequences of LLVM and finally translated into JSON format (an open standard file format) to target a BMV2 switch (a software P4 switch).

In [12], the authors used the P4 code generated by Whippersnapper, which is a P4 benchmark suit designed to study the impact of the compiler on performance to demonstrate that P4LLVM has exceeded p4c with respect to percentage increase in average latency versus number of operations in the action block and number of tables.

At present, p4c only has implementations of dead state elimination, constant propagation, constant folding and expression simplification [12]. In contrast, the LLVM framework has been carefully designed and many other optimizations have been added, including all p4c characteristics. Therefore, the P4LLVM compiler will help target many common backends with minimal effort. Figure 3 describes the workflow on how to combine p4c and P4LLVM.

3.4. Multi-target compiler - T4P4S

Another aspect of designing a compiler is targeting different hardware with a single compiler because it is much more efficient than creating a separate compiler just for a specific device. Vörös et al. proposed a multi-target compiler T4P4S (Translator for P4 Switches) in [14], which achieves a good balance between complexity, portability and performance. In the design process of T4P4S, the following principals are regarded as the main features of T4P4S: 1) This compiler should be retargetable, which means it can be easily deployed on another hardware; 2) The compiler should present a universal performance on all supported hardware and software and behave comparably to the native methods; 3) When processing forwarding logic to suit the target, the compiler should generate high level programs that do not require additional modifications. In order to meet these three objectives, the compiler T4P4S is separated into two parts: a hardware-independent core and a NetHAL (Networking hardware abstraction layer) responsible for the hardware-related parts. T4P4S currently only supports the original $P4_{14}$ language; new version extensions that support the new $P4_{16}$ language are still under development. Figure 4 shows the workflow of T4P4S.

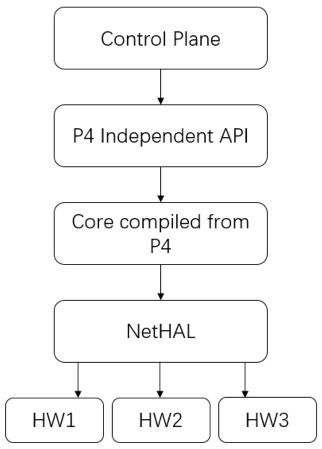

Figure 4: Architecture of T4P4S workflow [14]

3.5. Linux kernel targeted compiler - p4c-XDP

p4c-XDP is a Linux kernel target compiler, which can convert P4 programs into C code, then compile it to eBPF and then load it into the Linux kernel for packet filtering.

eBPF is an extended version of BPF and was reformed based on BPF by Alexei Starovoitov in 2013 [15]. BPF, known as Berkeley packet filter, was originally proposed by Steven McCanne et al. in [16]. Its purpose is to provide a method of filtering data packets and avoid useless copying of data packets from kernel space to user space. It initially consisted of a simple byte-code that is injected into the kernel from user space, where it is checked with a checker to avoid kernel crashes or security issues and attached to a socket and then runs on each received packet. In contrast, eBPF added new features to improve its performance, such as mapping and tail calls, and also rewrote the just-in-time compiler (a compiler can convert BPF instructions into native code). The new language is closer to the native machine language than before. Also, new attachment points will be created in the kernel.

An XDP program is a special case of the eBPF program and is used to process network packets. It is attached to the lowest level of the networking stack [17] and it is also a new fast path. XDP is used in conjunction with the Linux stack and the BPF is used to make packet processing faster.

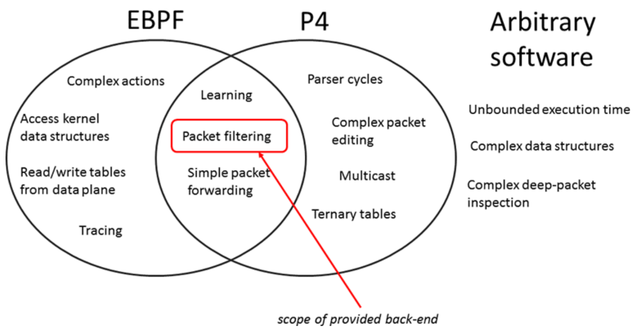

Figure 5: overlap between eBPF and P4 [18]

Although eBPF and P4 are two programming languages with different expression capabilities, there is overlap between the domain of network packet processing. Therefore, it is worth to develop a connection between P4 and eBPF [18].

Figure 5 presents the overlap between eBPF and P4.

3.6. Other compilers

Apart from the P4 compilers mentioned above, there are other P4 compilers with different functions, such as p4c-graphs, which can be used to generate visual representations of a P4 program [9]; p4test is a source-to-source P4 translator which can be used for testing, learning compiler internals and debugging [19]; p4c-ubfp can be used to generate eBPF code running in user-space [9]. However, due to insufficient research regarding these compilers, they cannot be introduced in detail in this survey, but they may become the motivation for future research.

4. Conclusion and future work

This survey outlines the latest research on P4 compilers from four different directions: The first kind is a hardware-specific compiler, which focuses on improving the efficiency of converting P4 programs into certain target hardware language like P4FPGA. The second type attempts to use existing mature compiler frameworks to optimize the performance of total packet processing and leads to an upgraded version of the standard P4 compiler. The third is to implement a multi-target compiler to reduce the effort spent on configuring with different hardware terminals like T4P4S. The last kind targets to the Linux kernel based on the standard p4c compiler.

In addition to the research effort on the P4 compiler, many researchers are also developing other extension features of P4. As mentioned at the beginning, SDN and P4 and other similar technologies will dominate the future network developing due to their flexibility and portability, which is a determining point compared to the traditional networking technology.

The P4 language and its compilers are still in the development stage. It is foreseeable that there will be more powerful P4 compilers in the future. These compilers can make the connection between the P4 program and the target hardware/software more robust and simple.

References

[1] H. Kim and N. Feamster, "Improving Network Management with Software Defined Networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[2] A. Gelberger, N. Yemini, and R. Giladi, "Performance Analysis of Software-Defined Networking (SDN)," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2013, pp. 389–393.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[4] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 122–135.

[5] J. S. da Silva, T. Stimpfling, T. Luinaud, B. Fradj, and B. Boughzala, "One for All, All for One: A Heterogeneous Data Plane for Flexible P4 Processing," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 440–441.

[6] p4language, "P4 compiler for the behavioral model," https://github.com/p4lang/p4c-behavioral.html, 2017, accessed June 11, 2020.

[7] ——, "p4-hlir," https://github.com/p4lang/p4-hlir.html, 2017, accessed June 11, 2020.

[8] T. P. L. Consortium, "$P4_{16}$ Language Specification," https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2017, accessed June 11, 2020.

[9] p4language, "$P4_{16}$ reference compiler," https://github.com/p4lang/p4c.html, 2020, accessed June 11, 2020.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[11] R. S. Nikhil and K. R. Czeck, "BSV by Example," *CreateSpace, Dec*, 2010.

[12] T. K. Dangeti, R. Upadrasta *et al.*, "P4LLVM: An LLVM Based P4 Compiler," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 424–429.

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[14] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[15] A. Starovoitov, "tracing filters with bpf," https://lkml.org/lkml/2013/12/2/1066/, 2013, accessed June 3, 2020.

[16] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture." in *USENIX winter*, vol. 46, 1993.

[17] W. Tu, F. Ruffy, and M. Budiu, "Linux Network Programming with P4," in *Linux Plumbers' Conference 2018*, 2018.

[18] p4language, "ebpf backend," https://github.com/p4lang/p4c/tree/master/backends/ebpf/, 2020, accessed June 3, 2020.

[19] O. N. Foundation, "PTF-based data plane tests for ONOS fabric.p4," https://github.com/opennetworkinglab/fabric-p4test/, 2020, accessed June 3, 2020.