Poster : Loading Programmable Data Plane Programs to Virtual Plane

YuXin Zhao
Tongji University
Email: zoe.yx.zhao@gmail.com

***Abstract*—Virtualization of the programmable data plane allows multiple virtual pipelines to be placed on the same physical programmable device, enabling more flexible network function composition, debugging, etc. Existing proposals realize virtualization with a hypervisor-like program to emulate users' programs, which becomes the mainstream of the current methods. In spite of the progress achieved, their designs lack study of how to load other programs on this hypervisor. In this poster, we present HyperC, the first compiler for virtualization in programmable data plane, which helps to build a complete virtualization system. HyperC specially optimizes its IR, which makes the hypervisor acquire a decreasing delay by 26.3% on average. At the same time, we solve the placement problem of different users under the restriction of virtual plane resources.**

I. Introduction

Based on the reconfigurable match-action architecture[1], [2],the Programmable Data Plane(PDP) enables network operators to customize the behaviors of network devices which used to be functionally fixed. With the help of domain specific languages like P4[3],NPL[4], more and more Network Functions (NFs) start to be offloaded into network devices which has been proved that can benefit from both high performance and spare more CPU power for user applications in servers.

However, some inherent limitations of the PDP hinder the trend of offloading NFs into PDPs. (1) The existing PDP can only be customized by one PDP program so it is incapable of meeting the requirements of multi-tenancy. (2) Dynamically reconfiguration of the device is impossible. The data plane has to be interrupted for reloading new program. And it is hard to maintain the consistency of network states.

Virtualization is one of the solutions to break those limitations. At the moment, the mainstream and hardware-platform-independent PDP virtualization approach is to provide a "hypervisor" program[5], [6], which can simulate other programs through its modular abstraction of the switch pipeline. Multiple user programs can be "loaded" on it by translating to its inner table entries. These entries can be injected by the control plane and matched by packet headers at running time to choose the target actions . Re-configuring these "upper" user programs actually requires only updating corresponding table entries without interrupting the data plane.

However, existing works in this direction mainly focus on the design of this hypervisor PDP program, but do not study or realize the compiler required to translate and load other user programs to the hypervisor. They manually compile the programs used for their experiments. But, realizing this compiler is not just need to code this manual process.

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

Translating and loading user PDP programs to the hypervisor poses the following research challenges:(1) Translating high-level language programs into table entries of the hypervisor. (2) Optimizing the translated table item structures to improve the hypervisor efficiency. (3) Placing different users' programs reasonably according to their requirements under the condition of limited virtual plane resources.

In this poster, we propose HyperC, which is the first work to study the compiler for virtualization in programmable data plane(VPDP). We use "assignment merging" to optimize the translated user programs, greatly reducing the actions to be executed by the hypervisor, which improves its work efficiency. At the same time, we mathematically model the placement problem of different user programs into an integer linear programming problem to make it have the optimal solution.

II. The Workflow of HyperC

A. Converting AST to DAG

The workflow of HyperC is divided into three main steps. The first is to convert the program AST into a simpler DAG by adding a front-end phase to the PDP language compiler[7]. The node of the DAG represents a match-action unit of the original program, and the link represents dependency between them. Match-action units, i.e. tables, are the main components of a PDP program. Packets header matches on these tables' entries and do actions. The execution order and logical branch statements determine the dependencies between them. At the same time, the main body of the hypervisor is a series of identical special-designed match-action units (named "slot"). These special match-action units enumerates all the action primitives for simulating arbitrary user programs' writing. We first convert this DAG into a uniform linear sequence by using the topological sorting algorithm. Then its node can be mapped to a slot of the hypervisor and its inner data are translated to the slot's table entries. Thus, a user program can be loaded to the virtual plane.

B. Assignment Merging

The second step is to use assignment merging to optimize the content in the DAG node.

We observe that assignment actions account for the vast majority in general programs and have the possibility of being optimized for their translated forms.

According to the sample programs provided by P4 tutorial, the average proportion of assignment actions to all actions is 53% and the average number of assignment statements in one logical stage is 2.64 which means in many cases, one match table will call multiple assignments. Without optimization, these assignments have to be mapped to different slots in the hypervisor because action primitives enumerated in each slot are limited. The more slots are used, the lower the performance will be, and the slots' remaining quantity may not be enough for the rest of the user program.

At the same time, in order to adapt to arbitrary programs, all users' packet headers and metadata definitions have already mapped to a whole data bus in the hypervisor program that the hypervisor should lookup some special tables to distinguish them. Then just like many IO optimization methods used in the computer system, data assigned in one logical stage does not actually need to be partitioned for their original meaning but can be seen as a whole block and be processed only once.

In our HyperC, we select variables operated only by one DAG node while the operation is assignment action. Then we construct a new variable combining their bit widths as the variable of actual deployment. Due to the restrictions we have imposed, this merge operation will not increase table entries which aren't needed originally, but simply reduce the operations required by the hypervisor.

C. Multiple User Pipeline Placement Problem

The third step is to output the placement scheme of multiple user programs on the hypervisor.

We model this problem as an integer linear programming problem. At a certain moment, there are M users' programs: $P_1, P_2 ... P_m$. For users' program i, it has nbU_i match-action units and for one of its match-action unit u, it needs $r_{i,u}$ resources in the hypervisor and can be assigned to use x resources of slot s, we denote it as $x_{i,u,s}$. Our first objective is to minimize the number of recirculations T. We define that the hypervisor has N slots in real but infinite slots in logical that the slots with the same result of module N belongs to the same real slot and share the resources R . (Through recirculation, the customer program may still lack resources, and we will say that the hypervisor can not support deployment at this time). Let χ_i denotes the max number of slot id assigned to program i. Then our **objective** is : $min \sum_{i=1}^{M} \lceil \chi_i / N \rceil$ where for any program i : $if\ x_{i,u,s} > 0, \chi_i \geq s$

When the first objective is met, we further want to minimize the total number of used slots to improve the performance. This objective can be transformed as the following[8] $min \sum_{i=1}^{M} \sum_{u=1}^{nbU_i} \sum_{s=1}^{\chi_i} \lceil x_{i,u,s} / R \rceil$

Dependency Constraint : Like[8], we use boolean variable $D_{A,B}$ to indicate whether unit B behinds A in the linear sequence, and the start and end slot ids assigned for unit u are denoted as S_u and E_u. We have the dependency constraint :$\forall D_{A,B} > 0, E_A < S_B$

Assignment Constraint:All users' units must be assigned their required resources somewhere in the hypervisor pipeline, i.e. $\forall i, u, \sum_{s=1}^{\chi_i} x_{i,u,s} \geq r_{i,u}$

Capacity Constraint: For each real slot, it can not allocate more resources than it has, i.e. $\forall n \in \{0, 1, ..., N-1\}, \sum_{i=1}^{M} \sum_{u=1}^{nbU_i} \sum_{s \in \{a | a \bmod N = n\}} x_{i,u,s} \leq R$

We can now use the integer linear programming tool to obtain the optimal solution.

III. Evaluation

We evaluate the performance improvement due to Assignment Merging. We use a server with 2×6 Intel i5-10400F 2.90Ghz cores and 16GB memory to run the experiment. The 4 test program are the same as HyperVDP uses in their experiments. Figure 1 shows that compared with HyperVDP, HyperC helps to reduce use of slots and tables therefore a decreasing delay by 26.3% on average is gained. More importantly, a complete VPDP system can be built with HyperC.

Table(slot) usage / Delay Time

	P4	HyperVDP	HyperVDP-HyperC
L2_switch	1 /0.8ms	4 (1) /1.1ms	4 (1) /1.1ms
Firewall	3 /1.1ms	8 (4) /1.6ms	6 (2) /1.3ms
Router	4 /1.2ms	16 (4) /1.9ms	8 (2) /1.4ms
ARP Proxy	4 /1.1ms	10 (3) /1.8ms	9 (3) /1.4ms

Fig. 1. Comparison of table/slot usage and delay time.

References

[1] 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," vol. 43, no. 4, p. 99–110, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[2] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "Drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–14. [Online]. Available: https://doi.org/10.1145/3098822.3098823

[3] The p4 language consortium. [Online]. Available: https://p4.org

[4] Npl 1.3 specification. [Online]. Available: https://github.com/nplang/NPL-Spec

[5] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," ser. CoNEXT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 35–49. [Online]. Available: https://doi.org/10.1145/2999572.2999607

[6] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019.

[7] $P4_16 reference compiler. [Online]. Available : https://github.com/p4lang/p4c$

[8] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 103–115. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose