Structure optimization and optimal pipeline placement in programmable data plane virtualization scheme

Yuxin Zhao

Tongji University Shanghai 200333 China

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 like HyPer4 and HyperVDP realize virtualization with a hypervisor-like program to emulate customers' programs, which becomes the mainstream of the current methods. Despite the progress achieved, their designs still have problems from inefficient structure to waste of resources. At the same time, they do not consider how to reasonably place the pipeline of different customers on the hypervisor which may cause unnecessary recirculations. So, in this paper, we present HyperSwitch, an efficient data plane hypervisor with several new techniques to optimize its structure design and reduce the resource consumption. We also model the multiple pipeline placement problem into an integer linear programming problem so that it can have the optimal solution. Compared to existing work, HyperSwitch acquires a decreasing delay by 26.3\% on average and avoids almost all happenings of recirculation.*

Keywords: *Structure, programmable*

1. Introduction

With the advent of reconfigurable match-action table architecture [4,5] and domain-specific languages like P4[2]and NPL [1], the Programmable Data Plane (PDP) enables network operators to customize the behaviors of network de-vices. Ordinarily, each programmable data plane represents a single networking context defined by one PDP program. In order to support diverse sets of customers or to flexibly compose virtual functions together for complex packet processing, in many cases operators might desire more than one context for a given network device even if this device has only one physical data plane. And one answer to this is virtualization.

The literature on PDP virtualization can be classified into1) compiler-based approach and 2) hypervisor-based approach.

Compiler-based PDP virtualization [10][8] aims to provide a compiler-level virtualization layer to support the concur-rent execution of multiple PDP programs which essential idea is to merge multiple PDP programs into one as shown in Figure 1. Therefore, if the customers want to update or change their PDP contexts, the data plane still needs to be interrupted and reconfigured. So, the compiler-based schemes are partial virtualization or pseudo-virtualization.

On the other hand, hypervisor-based approaches [6][9] introduce a special purpose program that provides a platform on which multiple PDP programs can be installed. An overall architecture for this kind of approaches is shown in Figure 2. A special designed PDP program, which is called as the hypervisor, is the real context running on the physical data plane(①). And the customers' programs and their table entries are all translated to the hypervisor's tables' entries by the VPDP compiler to make the hypervisor work(②). Be-cause these table entries can be dynamically modified by the control plane, one customer's update would not interrupt the whole system, hypervisor-based approaches are more acceptable solutions.

Hypervisor-based approaches support live reconfiguration and flexible network function composition. At the same time, it is suitable for all kinds of hardware platforms. How-ever, it is non-negligible that hypervisor-based virtualization comes with a performance penalty. If the delay of the virtual pipeline is too longer than the native pipeline, it will be im-possible to use it in the actual scene. So how to improve its efficiency as much as possible is the focus of design.

The designs of existing hypervisors are still insufficient in efficiency improvement. And they do not consider the multiple virtual pipeline placement problem. This lack of consideration will lead to unnecessary pipeline recirculations which will greatly reduce efficiency.

This paper provides a novel, efficient hypervisor as well as an optimal solution to the multiple virtual pipeline placement problem on it. Experiments show that we acquire a decreasing delay by 26.3% on average and avoids almost all happenings of recirculation.

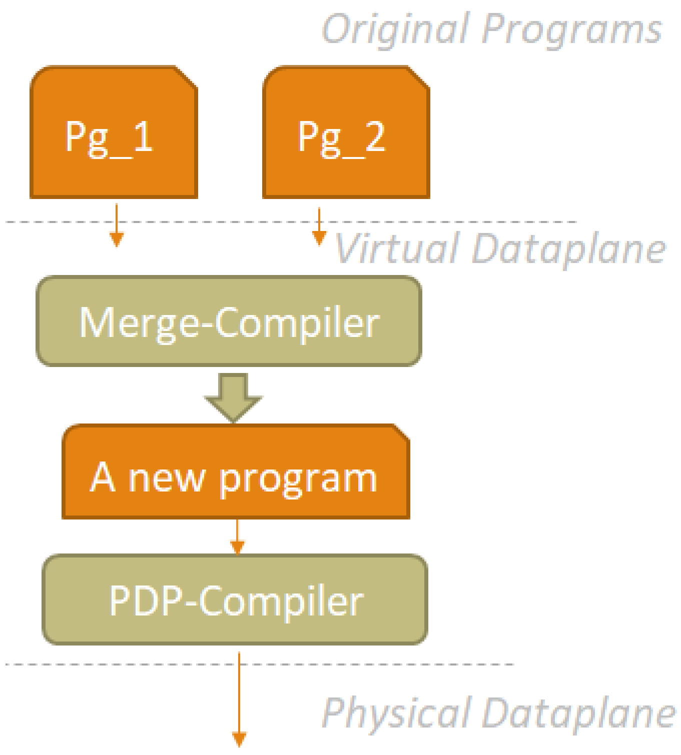


Figure 1 Workflow of compiler-based approach.

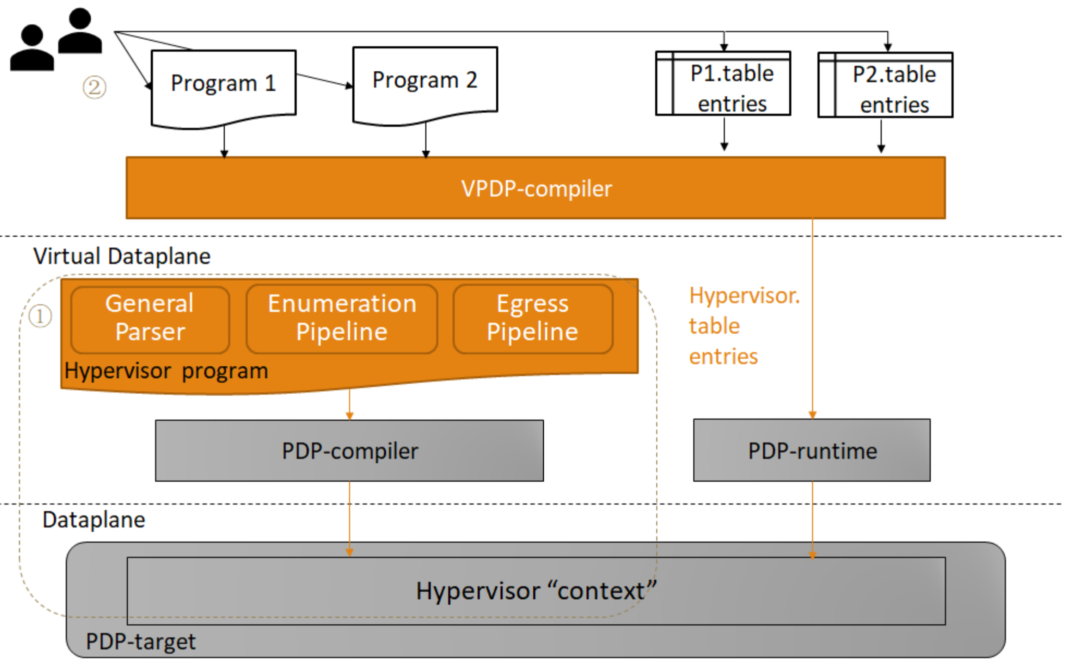


Figure 2 Architecture of hypervisor-based approach.

2. Background and Motivation

2.1 Background of Hypervisor-based Virtualization

The core part of a hypervisor-based virtualization framework is the hypervisor program. It is a special designed programmable data plane program that by modifying its table entries from the control plane, it can simulate other PDP programs like l2 switch written by the customers.

The hypervisor program is usually composed of three parts: the general parser, the enumeration pipeline and the egress pipeline. Figure 3 demonstrates this model.

To support virtualization, the general parser should be able to parse arbitrary header patterns. Since 1) the programmer can define arbitrary patterns of header struct. 2) The programmers' definition of the header struct can be gained from the VPDP compiler, the substance work of the general parser is to extract the whole packet header as a monolithic block of data according to its header length, and its inner contents can be identified by looking up tables in the later pipeline.

The enumeration pipeline emulates the customers' match-action pipeline. The match-action pipeline is the main part of a PDP program, although different PDP languages provide different language abstractions, this part can be modeled as a series of match-action units read and write fields in a data bus that the execution order of these units is determined by the program's control flow which is expressed as if-else statements. To emulate one logical match-action unit, the hypervisor needs to use multiple match-action units that form a "slot". The slot can be divided into the match part and the action part. To emulate arbitrary actions, the action part is an enumeration of all action primitives. And the hypervisor queries tables in the match part to identify customers and decide which actions to use. Such slots are arranged in sequence to form the enumeration pipeline of the hypervisor.

The egress pipeline builds, checks, sends, or drops the egress packet by calling functions provided by the PDP languages.

This three-part hypervisor design refers to the architecture of common PDP programs and is adopted by the former works, in which the specific data structure design and workflow design are different.

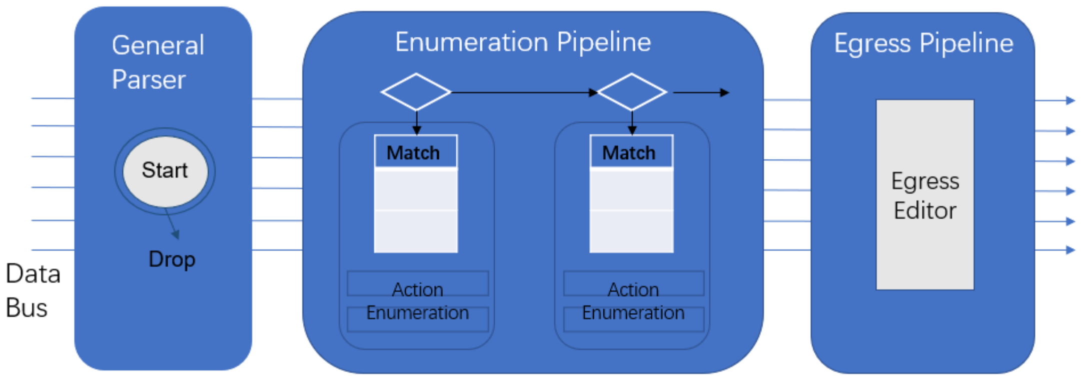


Figure 3 The hypervisor program model.

2.2 Motivation of HyperSwitch

We posit that a data plane hypervisor should satisfy two aspects of requirements: (1) It should reduce resource consumption and operation steps as much as possible to improve performance for emulating one program. (2) It should provide optimal placement under limited resources for multiple customer programs to improve the overall performance.

Although HyperVDP has improved in the first aspect compared with Hyper4, it still has many shortcomings. (1) The match-part of a slot consumes too many resources. Hyper4 designs a series of identical, possibly unused tables to repeatedly determine whether the complete match result has been gained and executed. HyperVDP uses bitmap to improve this, but it still uses the "ternary" match-kind to match on the whole data bus. To reduce the TCAM pressure, it classifies the data into three types: packet header, standard metadata and user-defined metadata, then uses three tables for each type and one table to combine their results to provide an action bitmap. All the same, the match field for this design is still very large, like matching on $dstAddr$ of the Ip header is evidently different from matching the whole packet header. (2) The action-part of a slot has not been optimized according to the actual situation. In practice, each action is used with different frequency, but neither of the former works considered this, resulting in a lot of unnecessary judgment processes. (3) Structure designed for emulating the control flow is rigid. The if-else statements express the control flow of the PDP programs, but Hyper4 does not support emulating these condition statements. HyperVDP designs a "condition binding" part for each slot which uses a dedicated set of tables to emulate the condition expressions. Its pipeline design is shown in Figure 4. But this design is rigid : 1) Each slot must go through these condition binding tables even if their contents are on the must path of the original program, 2) latter slots which contents are actually under the same condition states of the former slot cannot avoid experiencing these tables again.

All the above are the problems that our design will improve.

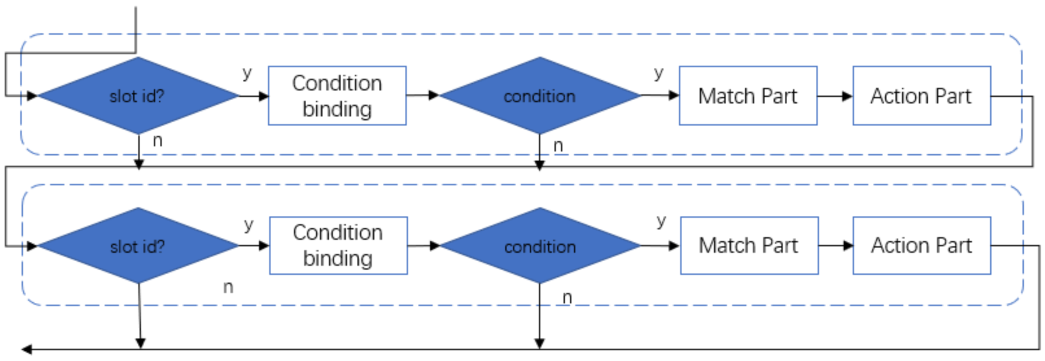


Figure 4 The enumeration pipeline structure of HyperVDP

For the second aspect, neither Hyper4 nor HyperVDP has considered this issue. Although HyperVDP proposes "Dynamic Stage Mapping", it is a theoretical method to mapping an unpredictable number of units onto the limited slots which is inspired by the mechanisms used in direct-mapped cache. However, it does not consider whether the mapping is necessarily successful or optimal. We can also find that simply using greedy algorithm or average allocation can not provide the optimal solution to this placement problem.

So in this paper, we present HyperSwitch, an efficient hypervisor model to support better PDP virtualization. In section 3 we introduce several new techniques to improve the design defects of previous works. And in section 4, we are the first to model the placement problem as an integer linear programming problem so that it can be solved.

3. The Hypervisor Program Design

This section introduces the new techniques designed to optimize different parts of the hypervisor program: grade header to reduce parser overhead, and grade bitmap, block assignment and decoupled control mapping for pipeline.

3.1 Optimizing General Parser

We define a novel hypervisor header to make the general parser work more efficient. The general parser needs to extract a special header which can express arbitrary length of header. The header struct here is named as "VPDP grade header".

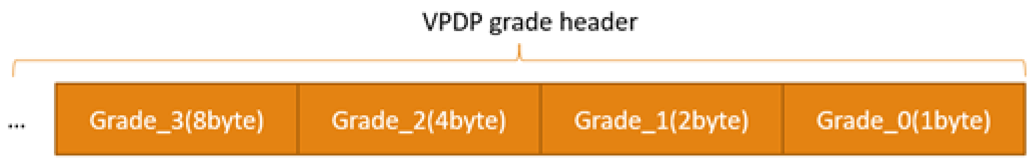


Figure 5 VPDP grade header.

The design draws lesson from the expression of binary numbers, that the whole structure is combined of a series units sized of 2^0 bytes, 2^1 bytes, 2^2 bytes and so on. In the parsing, the exact bit of the binary expression of the packet length determines whether this level of header units should be extracted. Therefore, arbitrary length of header can be extracted in one parser node by $O(\log n)$ judging times and equal or less extraction operations. While the former work, Hyper4, using a vector-like structure, its judging times and the number of extraction operations both reached $O(n)$.

After extracting the VPDP header, these header data will be moved to a desired place in a whole data bus defined by the hypervisor and be processed as part of the data bus by the later pipeline.

3.2 Optimizing Enumeration Pipeline

Compared with the existing works, we design a more concise slot structure. Figure 6 demonstrates this "locate-execute" structure. The slot reads and writes the data bus in which the hypervisor uses a whole block of metadata (named as VPDP user bus) to represent all kinds of customer-defined data. Although it can match on the whole metadata, this process obviously causes too much burden. So, we divide tables into two types: the location table and the execute table.

The match-location table sets the needed address number to the VPDP control bus, then the match-execute table uses these addresses to locate the data in the user bus to be matched and sets the actions need to be invoked. The action-location table sets the data addresses for locating the actions' inlet parameters, and according to the match result, the hypervisor invokes corresponding actions in the enumeration of actions. At present, this design declares the least number of tables.

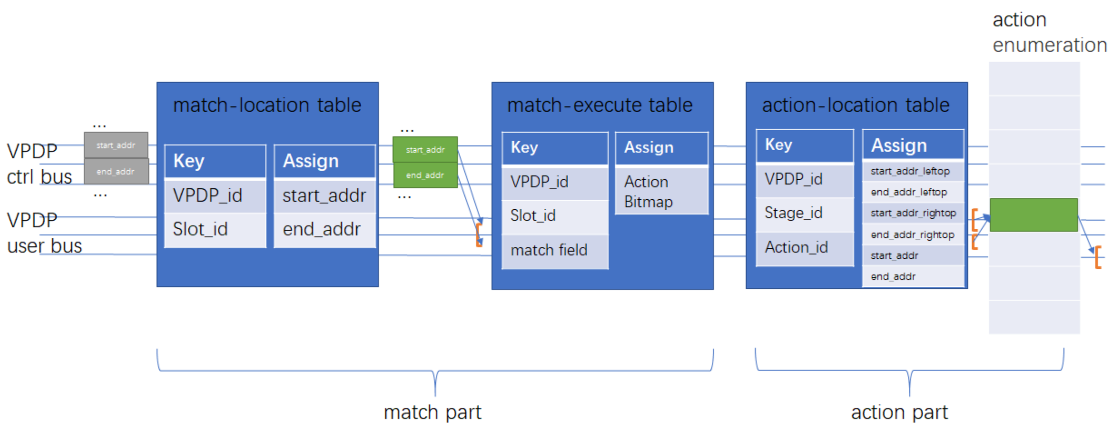
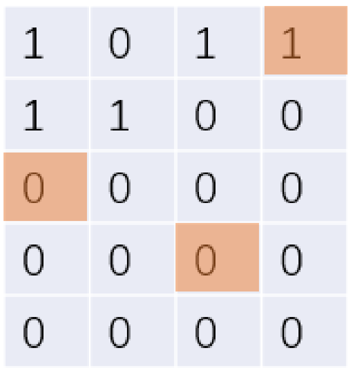


Figure 6 The "locate-execute" slot model.

3.2.1 Grade Bitmap

We improve the action bitmap design by grading these actions. For more practical use, we can observe that the use frequency for different action primitives is different. According to the 8 sample programs provided by P4 tutorial\cite{p4tut}, the average proportion of assignment statements to all action statements is 53\%, of add/subtract is 18\%, of drop is 23\%, and of other is 7\%. Therefore we can divide these action primitives into different batches according to their use frequency and types. The bitmap structure does not need to be modified for this design that only some bits in the bitmap are now used for whole batch judgement (figure 7). When selecting actions, the hypervisor first checks the whole batch position, if it equals to one, then checks the following bits for this batch. Under this grade bitmap design, about 80 percent of judgements can be avoided in the action part.



1	0	1	1
1	1	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 7 Grade bitmap, the orange bit stands for a batch.

3.2.2 Block Assignment

We further optimize the simulation process of the assignment actions.

Assignment action dominates the customers' PDP programs. According to the 8 sample programs provided by P4 tutorial, the average proportion of assignment statements to all action statements is 53% and the average number of assignment statements in one logical stage is 2.64.

2.64 means in many cases, one match table will call multiple assignments that in the former hypervisor design, these assignments must be mapped to different slots since action primitives enumerated in each slot are limited. One solution to alleviate this problem is to settle more assign primitives in one slot. However, this method is still not flexible enough.

We observe that since all customers' packet header and metadata definition have already mapped to a whole data bus in the hypervisor program that the hypervisor should look up the location table 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 and be processed only once.

Figure 8 shows an example of this block assignment process. The customer defines three kinds of data which will be assigned separately in one match-action unit in the original program. Without block assignment, the hypervisor needs to use three slots to emulate, however the VPDP compiler can allocate a continuous space for these three data, the location table now provides the address of this continuous space and only one assignment to this space is enough.

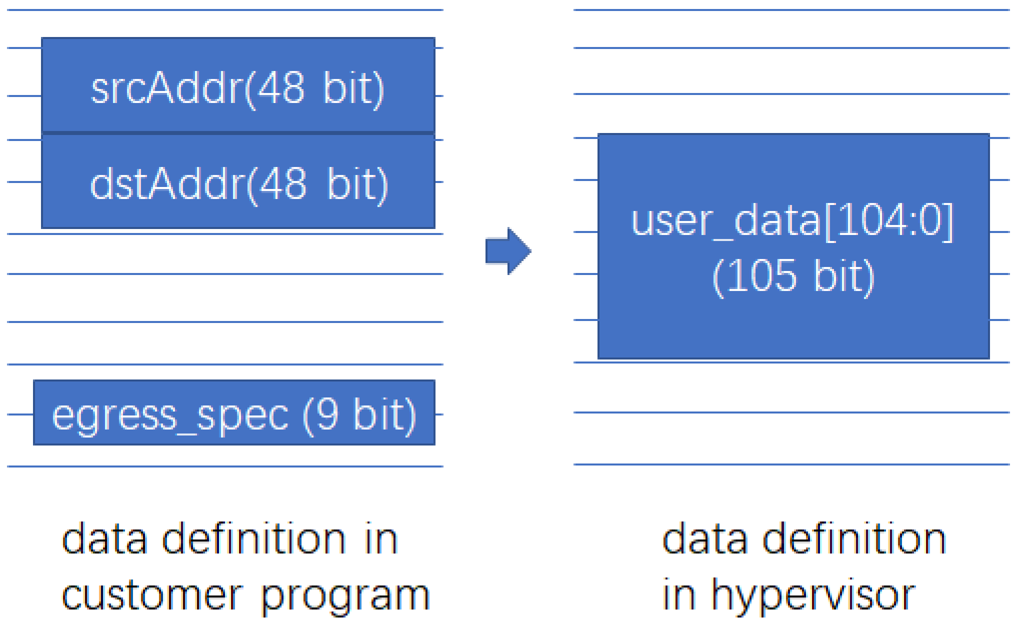

srcAddr(48 bit)
dstAddr(48 bit)
egress_spec (9 bit)
user_data[104:0] (105 bit)
data definition in customer program
data definition in hypervisor

Figure 8 Example of block assignment.

3.2.3 Decoupled Control Mapping

Our design decouples the tables for branch expressions (which we call condition branch node) with the match-action slots. To be specific, the hypervisor still uses a dedicated set of tables to emulate the branch expressions which match result is a stage bitmap, a branch result bitmap, and a $next\_id$ to mark when is the next time to call this condition branch node. When entering the enumeration pipeline, the hypervisor first checks the $next\_id$ initial value to decide whether to call this condition branch node for some customer programs do not use if-else statement, once these branch node tables are looked up, the hypervisor itself uses if-else statement to check the stage bitmap to decide whether this slot should be jumped, or which branch of the original program should be emulated. Thanks to the $next\_id$, the hypervisor can avoid looking up these tables for every slot. And these tables can be designed to support complex branch expressions. The structure for this design is shown in figure 9.

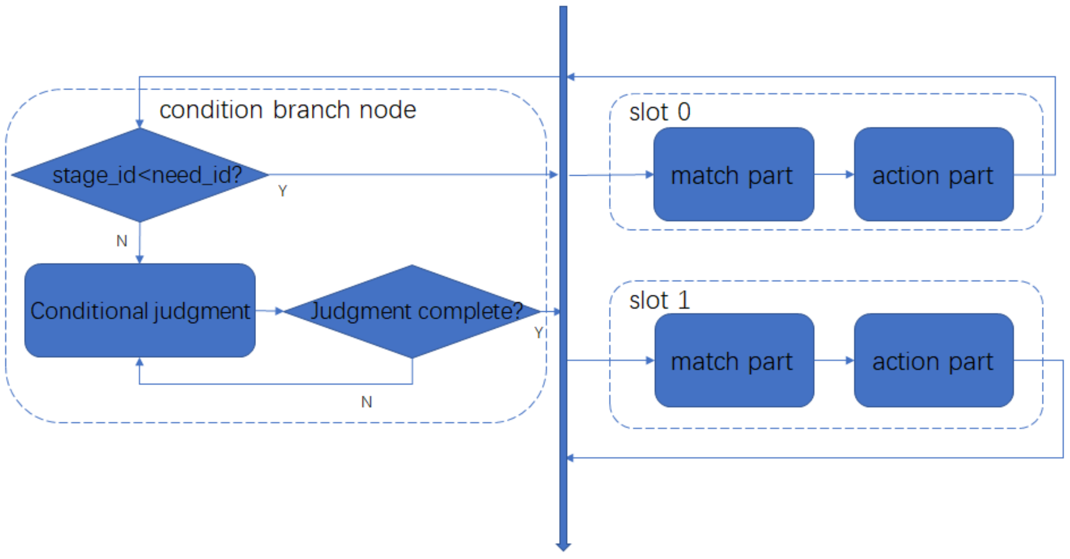


Figure 9 Decoupled condition branch node.

4. Multiple Virtual Pipeline Placement Problem

We are the first to solve the placement problem of different customers' pipelines under the condition of limited resources by modeling it as a multi-objective integer linear programming problem.

The pipeline of an ordinary PDP program can be transferred to a DAG where each node denotes a match-action unit and each edge represents the dependency between them. Then we can use the topological sorting algorithm to convert this DAG into a uniform linear sequence so as to determine which slot to present which unit. Suppose that the resources available in each slot are infinite, then we just need to load these units one by one. Although this assumption is adopted by previous works by default, it does not hold in reality.

We use a simple example to illustrate this problem. Assume that our hypervisor has 4 slots which each has R resources. Customer $program_1$ has 2 units which each needs R resources in the hypervisor and customer $program_2$ has 4 units which each needs $0.5R$. If we use greedy algorithm (i.e. first-come-first-serve algorithm) that allocates the first two slots to meet $program_1$ requirements then there is no enough slot for $program_2$ which then needs to be recirculated. This is because a slot's workflow is fixed, so it is impossible to deploy multiple units into one slot in one process. The better solution to this example is to load the unit of $program_1$ into two slots which leaves resources for $program_2$. But if we take the way of average allocation to ensure the resources of each program, it is not necessarily optimal. For instance, if $program_2$ is the same as $program_1$, then one of the best plans is to allocate the first two slots to $program_1$ and the next to $program_2$. So we find that we need to model this problem as an integer linear programming problem to solve it.

The hypervisor may know the customers' programs by loading entries from the control plane at one time, or these table entries who present a customer program can be inputted at different time. Because the hypervisor supports live reconfiguration, we can discuss the latter as the former. At a certain moment, there are M customers' programs: ${P_1,P_2...P_m}$. For customers' 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 , for recirculation will greatly affect the delay and throughput. 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 :

\begin{equation}
min \sum_{i=1}^M{\lceil \chi_i / N \rceil} \label{objective1}

\end{equation}

where for any program i :
\begin{equation}
if\ {x_i_,_u_,_s}>0 , \chi_i \geq s
\end{equation}

When objective \ref{objective1} 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:

\begin{equation}
min \sum_{i=1}^M\sum_{u=1}^{nbU_i}\sum_{s=1}^{\chi_i}{\lceil x_i_,_u_,_s/R\rceil}
\end{equation}

Dependency Constraint : Like\cite{C2ReSw}, 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 :

\begin{equation}
\forall D_A_,_B>0, E_A<S_B
\end{equation}
Assignment Constraint:

All customers' units must be assigned their required resources somewhere in the pipeline.

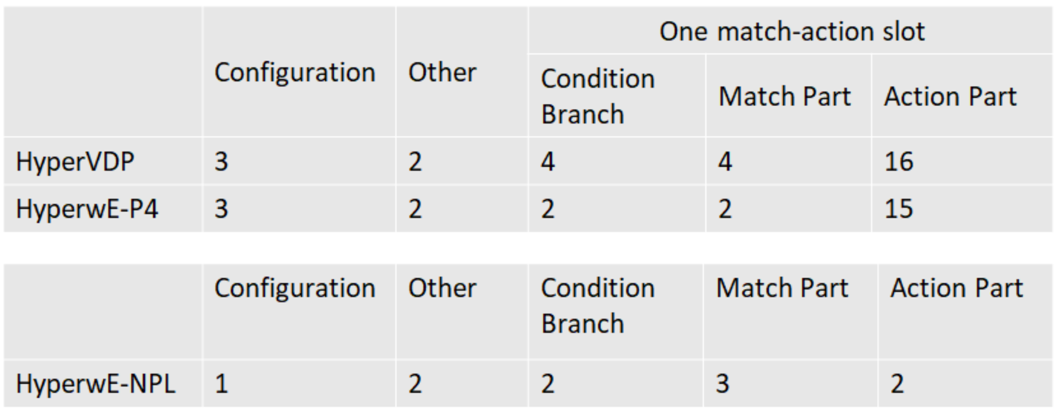
\begin{equation}
\forall i,u , \sum_{s=1}^{\chi_i}{x_i_,_u_,_s}\geq r_i_,_u
\end{equation}
Capacity Constraint: For each real slot, it can not allocate more resources than it has.

\begin{equation}
\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
\end{equation}

By modeling this placement problem, we can now use the integer linear programming tool to obtain the optimal solution.

5. Evaluation

We implement two versions of HyperSwitch program, one for P4 and the other for NPL. Figure 10 illustrates the table declare numbers for each version, which also uses HyperVDP(implemented by P4) as a control. Note that NPL provides "multiple lookup" feature so that tables in different slots can be the same. Our "locate-execute" model requires the least number of tables in one slot which means we needs the least operation steps and resources for declaration.

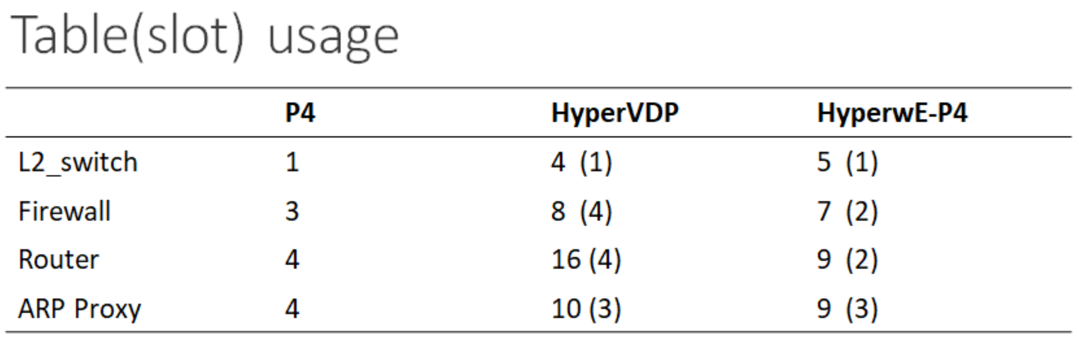


	Configuration	Other	One match-action slot		
			Condition Branch	Match Part	Action Part
HyperVDP	3	2	4	4	16
HyperwE-P4	3	2	2	2	15

	Configuration	Other	Condition Branch	Match Part	Action Part
HyperwE-NPL	1	2	2	3	2

Figure 10 Comparison of table declare numbers.

We evaluate the performance of BMv2-target HyperSwitch by comparing with HyperVDP and native P4-specific data plane. 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 11 shows that compared with HyperVDP, our design consumes less slots and tables which main reason is we use "block assignment". Also, the grade bitmap helps our hypervisor reduce 81\% judge times in the action part. Figure 12 illustrates the delay time. The increasing delay of HyperVDP ranges from 37.5\% to 63.6\% when comparing with native P4. For HyperSwitch, the increasing delay remains below 37.5\%, and the minimal increasing delay is only 16.6\%. When making a comparison between HyperSwitch and HyperVDP, we can see that HyperSwitch acquires a decreasing delay by 26.3\% on average.


Table(slot) usage

	P4	HyperVDP	HyperwE-P4
L2_switch	1	4 (1)	5 (1)
Firewall	3	8 (4)	7 (2)
Router	4	16 (4)	9 (2)
ARP Proxy	4	10 (3)	9 (3)

Figure 11 Comparison of table/slot usage.

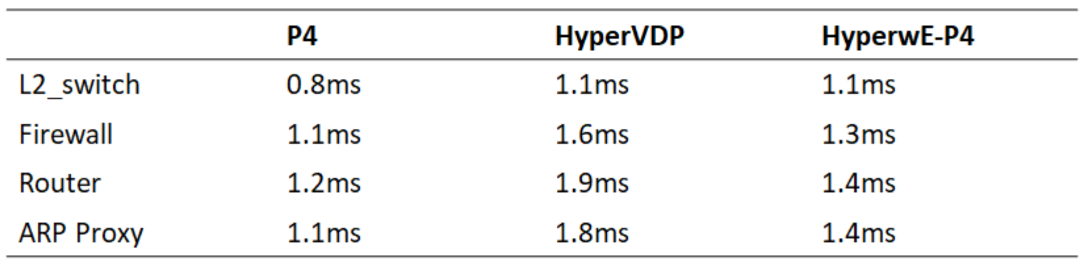

	P4	HyperVDP	HyperwE-P4
L2_switch	0.8ms	1.1ms	1.1ms
Firewall	1.1ms	1.6ms	1.3ms
Router	1.2ms	1.9ms	1.4ms
ARP Proxy	1.1ms	1.8ms	1.4ms

Figure 12 Comparison of delay time on p4-bmv2 platform.

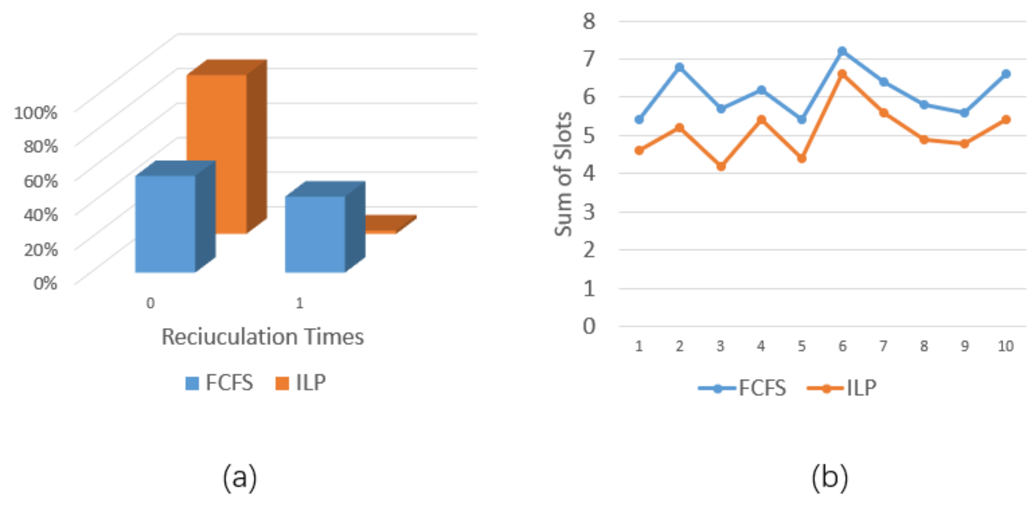


Figure 13 Comparison of recirculation times and sum of used slots.

For multiple pipeline placement, we compare the improvement of integer linear programming algorithm with first-come-first-serve algoritm used by the HyperVDP. We randomly assign the resource requirements of two random customer programs and check the placement result of the two algorithms. Figure 13(a) shows that the linear programming algorithm avoids most of the recirculations. And figure 13(b) compares the average used slots in ten groups of programs. The multi-objective integer linear programming also reducing the slot numbers by 16.46\% on average.

6. Conclusion

In this paper, we present HyperSwitch, an efficient hypervisor for better virtualization of the programmable data plane and an optimal solution to the multiple pipeline placement problem. In HyperSwitch, we propose several innovative techniques such as VPDP grade header, the "locate-execute" slot model, the grade bitmap, the block assignment, and the decoupled condition node. Evaluations show that we further reduce the resource resource consumption and latency comparing with former works. And the multi-objective integer linear programming algorithm, compared with first-come-first-serve, can avoid almost all happenings of recirculation.

This paper mainly focuses on the core part of the hypervisor-based VPDP framework: the hypervisor program. In our future work, we will study the VPDP compiler design and build a complete VPDP framework.

References

[1] [n.d.].NPL 1.3 Specification. https://github.com/nplang/NPL-Spec
[2] [n.d.].The P4 Language Consortium. https://p4.org
[3] [n.d.].P4 Tutorial. https://github.com/p4lang/tutorials
[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McK-eown, Martin Izzard, Fernando Mujica, and Mark Horowitz.(2013). Forwarding Metamorphosis: Fast Programmable Match-Action Pro-cessing in Hardware for SDN.43, 4 (Aug. 2013), 99–110. https://doi.org/10.1145/2534169.2486011
[5] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Var-gaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. (2017). DRMT: Dis-aggregated Programmable Switching. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (Los Ange-les, CA, USA)(SIGCOMM '17). Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3098822.3098823
[6] David Hancock and Jacobus van der Merwe. (2016). HyPer4: Using P4to Virtualize the Programmable Data Plane (CoNEXT'16). Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/2999572.2999607
[7] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. (2015).Compiling Packet Programs to Reconfigurable Switches. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). USENIX Association, Oakland, CA, 103–115. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose
[8] Hardik Soni, Thierry Turletti, and Walid Dabbous. (2017). P4Bricks: Enabling multiprocessing using Linker-based network data plane ar-chitecture. (Nov. 2017). https://hal.inria.fr/hal-01632431 workingpaper or preprint.
[9] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. (2019). HyperVDP:High-Performance Virtualization of the Programmable Data Plane.IEEE Journal on Selected Areas in Communications37, 3 (2019), 556–569.https://doi.org/10.1109/JSAC.2019.2894308
[10] Peng Zheng, Theophilus Benson, and Chengchen Hu. (2018). P4Visor: Lightweight Virtualization and Composition Primitives for Buildingand Testing Modular Programs. In Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (Her-aklion, Greece) (CoNEXT'18). Association for Computing Machinery,New York, NY, USA, 98–111. https://doi.org/10.1145/3281411.3