P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture

Hardik Soni, Thierry Turletti, Walid Dabbous

P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture

Hardik Soni, Thierry Turletti, Walid Dabbous

Abstract

Packet-level programming languages such as P4 usually require to describe all packet processing functionalities for a given programmable network device within a single program. However, this approach monopolizes the device by a single large network application program, which prevents possible addition of new functionalities by other independently written network applications.

We propose P4Bricks, a system which aims to deploy and execute multiple independently developed and compiled P4 programs on the same reconfigurable hardware device. P4Bricks is based on a Linker component that merges the programmable parsers/deparsers and restructures the logical pipeline of P4 programs by refactoring, decomposing and scheduling the pipelines' tables. It merges P4 programs according to packet processing semantics (parallel or sequential) specified by the network operator and runs the programs on the stages of the same hardware pipeline, thereby enabling multiprocessing. This paper presents the initial design of our system with an ongoing implementation and studies P4 language's fundamental constructs facilitating merging of independently written programs.

1 Introduction

P4 is a high-level language for programming protocol-independent packet processors. It allows reconfiguring packet processing behavior of already deployed data plane hardware devices, introducing new protocols and their processing to the devices, hence decoupling packet processing hardware from software. This provides high degree of flexibility for programming new network applications and packet processing functionalities using reconfigurable hardware like RMT [1] and Intel Flexpipe™. With P4, a network operator can execute one program at a time on the target reconfigurable network device. However, as the number of features or network applications to be supported by the device

grows, P4 programs increase in complexity and size. The development and maintenance of such monolithic P4 programs containing all the possible features is error prone and needs huge time and effort as program complexity grows. On the other hand, this approach does not allow to easily compose independently written modules in a single P4 program. Network devices can have different packet processing requirements according to their role and location in the network. Deploying one large program for a small subset of applications results in inefficient utilization of the reconfigurable device resources.

P5 [2] optimizes resource utilization by leveraging policy intents specifying which features are required to remove excess applications and packet processing functionalities. However, with P5 there is still a large monolithic program to be configured based on the policy intents. ClickP4 [3] proposes a smart modular approach in which separate modules can be developed within the ClickP4 programming framework and then manually compiled in a large program. However, programmers are required to know the code of the ClickP4 library modules to integrate a new module into the ClickP4 framework as source code modifications may be required on the already developed code base. Basically, P5 allows removing extra modules and features from already composed small P4 programs, whereas ClickP4 gives choice to select from a list of modules. Most importantly, with both P5 and ClickP4, packet processing functionalities on a device can not be easily composed using independently developed and compiled P4 programs.

Hyper4 [4], HyperV [5] and MPVisor [6] propose virtualization of programmable data plane in order to deploy and run independently developed multiple P4 programs on the same network device at the same time. In these approaches, a general purpose P4 program working as a hypervisor for programmable data plane is developed, which can be configured to achieve functionally equivalent packet processing behavior of multiple P4 programs hosted by it. However, virtualization requires minimum 6-7× and 3-5× more match+action stages for every P4 program compared to its native execution for Hyper4 and HyperV, respectively. Also, such approaches show significant performance degradation for bandwidth and delay, thereby nullifying the benefit of high performance reconfigurable hardware.

Meanwhile, executing efficiently multiple P4 programs at a time on a same target device is highly desirable. We believe that a network operator should be able to easily add any features on its target device with programs potentially developed by different providers.

We present the design and architecture of P4Bricks, our under development system that aims to deploy and execute multiple independently developed and compiled P4 programs on the same reconfigurable device. P4Bricks comprises two components, called Linker and Runtime. The Linker component merges the programmable parsers and deparsers and restructures the logical pipeline of P4 programs by refactoring, decomposing and scheduling their match-action tables (MATs). The Runtime component translates the dynamic table updates from the control planes of different applications into the tables of the merged pipeline. P4Bricks merges and executes MATs of multiple compiled P4 programs on the stages of the same hardware pipeline, thereby enabling multiprocessing. The idea is to provide a seamless execution environment for P4 programs in a multiprogram environment without any changes required in its control interface and MATs definitions. With P4Bricks network operators can specify the packet processing policy on the target device in terms of compiled P4 programs and composition operators using a simple command line interface.

This report presents the initial design of our system with an ongoing implementation and studies P4 language's fundamental constructs facilitating merging of independently written programs. It is organized as follows. Section 2 provides an overview of the P4Bricks system. Section 3 describes Linker, which composes compiled P4 programs using the only knowledge of MATs definitions. Then, Section 4 describes the Runtime module that interacts with the Linker component and the control plane of P4 programs to manage flow entries in the MATs of the programs.

2 System Overview

In this section, we provide a brief overview of the P4 language and we then introduce our system, describing merging of programmable blocks at link time and their management at runtime.

2.1 P4 background

P4 is a high-level language, based on programmable blocks, which is used for programming protocol-independent packet processors. The programmable parser, deparser and logical match-action pipeline are the main programmable data plane blocks. Parser blocks are programmed by defining packet header types and fields, declaring instances

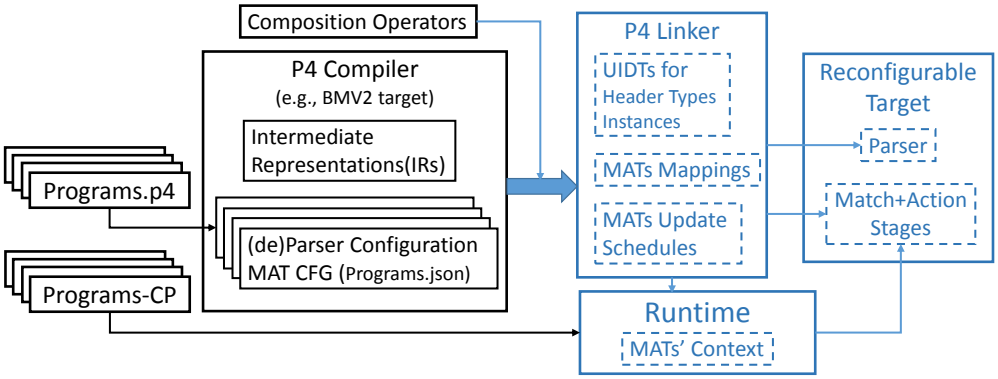


Figure 1: System Overview

of the types and defining Finite State Machine (FSM) to extract the header instances from the packets' bit streams. Match-action units are defined as tables (MATs) using match keys and actions. Match keys can be header fields or program's state variables and actions may modify and create header instances, their fields and hardware target specific fields. To create the control flow required for packet processing, programmers can use *if* and *switch* constructs to select next tables to process based on processing logic. Finally, packets are reassembled using the processed header instances by programming the deparser control block. Apart from programmable blocks, P4 provides APIs and a control interface to manage flow entries in MATs at runtime.

2.2 The P4Bricks System

P4Bricks consists of two components, Linker and Runtime. Linker takes as input the data plane configuration files generated by P4 compiler from source files of independently written P4 programs, see Figure 1. It merges the programmable parser, pipeline and deparser blocks defined in the compiled configuration files of P4 programs according to packet processing policy described using parallel and sequential composition operators, also given as input. Linker does not assume knowledge of flow entries in MATs of P4 programs while merging the pipelines at link time. On the other hand, Runtime processes the MATs flow updates from the control plane of P4 programs at runtime, in accordance with composition operators and traffic isolation enforced by the network operator.

In order to merge the parsers, Linker has to identify equivalent header types and header instances present in the different P4 programs to enable sharing of common

header types and instances. Two packet header types are said equivalent if they have the same format. Linker maps header types to Unique IDentifiers (UIDs) and stores the mappings between program specific IDs and UIDs in a table called header types UID Table (UIDT). Equivalent packet header instances are identified while merging the parse graphs by matching the sampling locations of parsers in the packet bit stream and the instances of header types extracted from the location. The mapping between program specific IDs and UIDs is also stored in a table called header instances UIDT. As merging parsers of two programs creates another parser, the composition operators can be recursively applied to merge parsers of any number of programs.

P4Bricks considers the packet header instances (extracted or emitted) and specific entities like interfaces and meters as *data plane resources*, which are shared and accessed for packet processing by the logical pipelines of the different P4 programs. Linker replaces program specific IDs of header types and instances given by the compiler with the mapped ones in UIDTs before merging pipelines according to the composition operators. This unifies different references (IDs and names) for equivalent header types and instances used in P4 programs, thereby identifying sharing of resources.

A deparser control block provides the *emit* sequence of header instances to reassemble the packet, as defined by the programmer. If the instance is valid, emitting a header instance will append the instance to the outgoing packet. Similar to parser, two deparser blocks can be merged if the P4 compiler generates a DAG of *emit* instances, which encodes all possible *emits* functions after appending a header instance. However, as the P4 compiler generates a list to append header instances, the partial order (providing relative location of appending the header instance) required to merge instances of different header types at the same level of the network stack can not be identified. In our current P4Bricks implementation, we use the merged parse graph to identify the partial order among the header instances to be compatible with current P4 specifications. We note that if a future version of P4 makes use of DAGs to represent deparser control blocks, this could allow conceptually correct merging of deparsers without dependence to the parser block.

The packet processing control flow in the compiled configuration file of a P4 program is commonly represented as a DAG with each node representing packet processing using a MAT. For each pipeline of a program, Linker decomposes the MATs control flow graph (CFG) by adding resources as nodes, splitting each MAT node into match

and action *control nodes*, and adding dependencies between control nodes according to resources accessed. Linker generates read-write *operation schedule* graph (OSG) for each resource from the decomposed CFG to capture all possible access orders and types (read or write) of operations executed on the resource due to packet processing control flow in the pipeline. Linker merges packet processing CFGs of all the P4 programs and the OSGs generated from them for each resource according to composition operators. Then, Linker refactors the MATs, regenerates the CFG and maps the refactored MATs to physical pipeline stages while respecting the merged read-write OSG for each resource, the MAT control flow of all the P4 programs and available physical match memory type and capacity in the stages. We introduce two concepts to facilitate this restructuring : 1) Vertically decomposing the MATs into sub MATs and 2) Performing out-of-order write operations in OSG of any resource. This allows mapping of a sub MAT on available physical match memory type (e.g., exact, ternary or Longest Prefix Match) in the pipeline stage, even if the complete MAT can not be scheduled to the stage due to control dependency. Apart from creating new MATs and CFGs, Linker produces the mappings between new MATs mapped to hardware stages and the MATs of all the P4 programs. It also prepares the MATs mappings and update schedules of the decomposed MATs that will be used by the Runtime component, as shown in Figure 1.

Runtime executes in the control plane of the target device and acts as a proxy to the control plane of P4 programs in order to manage their MATs defined in configuration files. It uses UIDTs and MAT mappings generated during linking to translate MATs update from the control plane of programs to the tables mapped to physical pipeline stages by Linker. Runtime is responsible for maintaining referential integrity across the sub MATs of a decomposed MAT to provide consistent MAT update. For every decomposed MAT with its sub MATs mapped to different stages, Runtime updates the entries of the sub MATs according to the schedule generated by Linker. Moreover, it regulates the flow updates from the control plane of all the P4 programs to enforce flow space isolation dictated by the network operator.

3 Linker

In this section, we describe the static linking process of compiled configuration files of two independently written P4 programs.

3.1 Merging Parsers and Deparsers

The parser block in P4 is modeled as a FSM, which encodes a directed acyclic parse graph. Each vertex represents a state and the edges describe the state transitions. Using *extract* construct of P4, each state can extract zero or more header instances by advancing the current index in the bit stream of the incoming packet according to the header type definition. The other fundamental *select* construct of P4 allows to specify lookup fields and value to program state transitions or identify next header types in the packet. Apart from *extract* and *select*, P4 provides other constructs namely *verify*, *set* and *lookahead* respectively for error handling, variable assignments and reading the bits in the stream beyond the current index without incrementing it. Essentially, merging of two parsers requires creating a union of their Directed Acyclic Graphs (DAGs) encoded as FSMs. However, as programs can be independently implemented, they may not use the same identifiers and names. So, it is necessary to identify the equivalent header types, instances and parse states defined in the parser blocks. In the following we define the notion of equivalence of header types, parse states and header instances between two programs and explain how to find the equivalence relationship. Then, we describe our method to merge the states and create union of DAGs and thereby FSMs.

3.1.1 Equivalence of Header Types

A header type is defined as an ordered sequence of fields with bit-widths. Two fixed-length header types are equivalent if their ordered sequences of bit-widths are the same. Regarding variable length types, the length of the variable bit-width fields must depend on the same fixed width field in the header types and their maximum lengths must be identical. The length indicator field is uniquely identified by its bit-width and its offset from the start of the header. Using these definitions, we create a UID for each header type as an ordered sequence of bit-widths corresponding to its fields. In case of a variable length type, a UID is created considering its maximum possible length and the identifiers of the length indicator field. Header types UIDT maintain the mapping between UIDs and program specific identifiers for all the header types in all the programs.

3.1.2 Equivalence of Parse States and Header Instances

Parse states extract the bit streams into instances of header types defined in the program. So, the equivalence of parse states and header instances are correlated. The instances extracted from equivalent parse states are mapped to each other in the instance UIDT. A parse state of a P4 program's parser is equivalent to a parse state of another program's parser if they satisfy the following conditions:

C1: both states extract bits from the same location in the bit stream of the packet. So, the current bit index points to the same location in the bit stream of the packet when the parsers visit the states.

C2: both states advance the current bit index by the same number of bits and extract the equivalent header types.

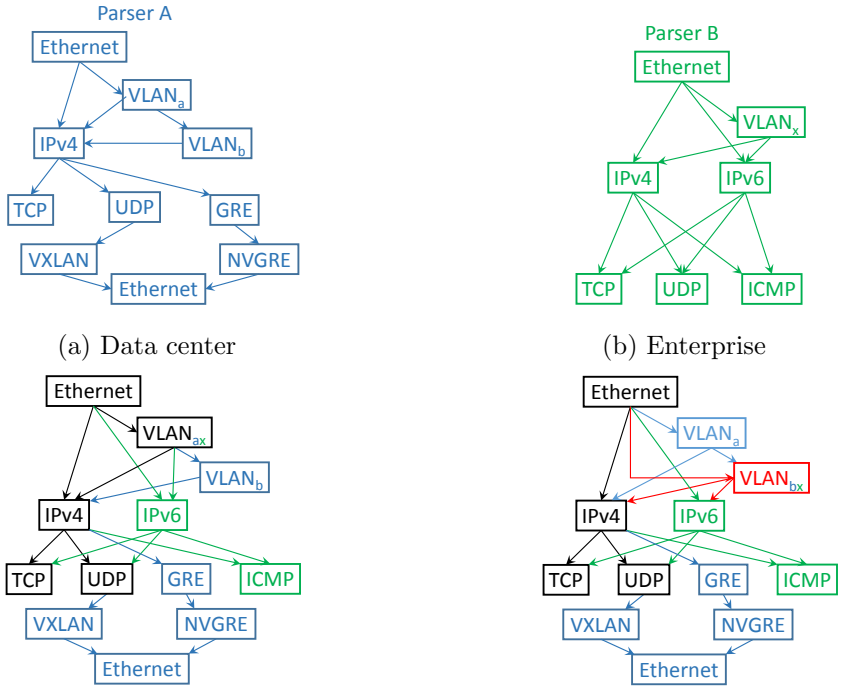
C3: if both states have *select* expression, then the lookup fields used in the expressions should be equivalent[1].

These conditions cover the scenario of *lookahead* construct also, where the states do not advance the current bit index but read the same set of bits to identify next transition states or store the value.

Let us take an example of merging two parse graphs of P4 programs to process Data center and Enterprise[2] network traffic, shown in Figures 2a and 2b, respectively. The parse graph of Data center has two VLAN states extracting double tagged headers in two VLAN instances a and b, whereas the parse graph of the Enterprise network has a single VLAN state extracting header into the single instance named x. As shown in Figures 2c and 2d, when merging the two parse graphs, $VLAN_x$ should be equivalent to $VLAN_a$ but not to $VLAN_b$. Indeed, only $VLAN_x$ and $VLAN_a$ states extract the bits from the same current bit in the topological order to satisfy condition C1. First, we select the set of nodes (i.e., states) having 0 in-degree at each iteration for each parser graph in order to traverse the parse graphs in topological order. Then, we find equivalence states by verifying C2 and C3 conditions for all possible pairs of states from the two sets. After that, we remove the 0 in-degree nodes with their outgoing edges and continue traversing.

[1] We assume all keyset expressions in *select* to be known values at compile time.

[2] The parse graphs are inspired from Figure 3 in [7].



(a) Data center (b) Enterprise

(c) Valid Merge (d) Invalid Merge

Figure 2: Merging parse graph of two parsers

3.1.3 Merging of Parse Graphs

We begin with merging the *select* transitions (i.e., edges in parse graphs) of two equivalent states by taking union of their keysets to create a single state. Regarding transition state of each *select* case, we find their equivalent state from the mappings and recursively merge them. If the two states have the same keyset, the corresponding transition states must be equivalent and we merge them too. We note that for a same keyset, two parsers can not transit to two different states. For instance, the value $0x0800$ of *EtherType* can not be used to transit to IPv4 state by one parser and to IPv6 state by the other. Allowing such ambiguous transitions creates non-deterministic FSM, resulting in a scenario where a packet can be parsed in different ways, which creates an ambiguity during packet processing. In the case of sequential processing, merging the parsers of two programs is not sufficient to find the equivalent header instances between them. Let us consider an example of chaining of encapsulation-decapsulation network functions, where the first program pushes new header instances and the second parses the pushed header instances to process traffic in the network. Executing them on the same target with sequential composition requires identifying the equivalence relationship between header instances.

For this purpose, we map the topological order of instances parsed from the merged parse graph to the sequence of emitted header instances in the deparser control block of the first program. We iteratively map 0 in-degree instances from the merged parser to the emitted instance from the deparser of the first program in sequence and removing instances from both. At each iteration, we search for an unmapped instance of merged parser having equivalent header type to the emitted unmapped instance. If there is any such instance pair, we create an equivalence mapping between them.

Apart from *select* and *extract*, parse states may *set* data plane resource and *verify* them for error handling. Header instance and its fields extracted from the equivalent states are the data plane resources shared between two states as well as programs. To merge two equivalent parse states, we consider *set* and *verify* P4 constructs as program statements performing *read* or *write* operations on the shared resource by the parse states. We consider local variable assignment statements in the states as read operations on shared resources and write on the non-equivalent data plane resource. We simply merge the program statements even though they may perform redundant operations (e.g., verifying checksum or fields' value). If either of the program is modifying the shared resources, we create a separate copy of the resources for the programs.

3.1.4 Traffic Isolation

We illustrate the need of traffic isolation in parsers with the example shown in Fig. 2. Parsed packet headers of double tagged VLAN traffic should be exclusively processed by the MATs of the Data center program and MATs of the Enterprise program should process VLAN traffic only if the packets have a single tag. Also, if both the programs have only TCP header fields based MATs, the IPv6 based TCP traffic should not be processed by the MATs of the Data center program. In order to provide traffic isolation emerging from parsers, we add a data plane resource as a metadata field with a bit width equal to the number of P4 programs to be merged. Each bit of this field indicates validity of the packet header instances for a program. In every state of the merged parse graph, we mask this field with a binary code that specifies the programs having equivalent state in their parser. For instance, in the example of Figure 2c, $VLAN_b$ and $IPV6$ will mask the indicator with $0b01$ and $0b10$, respectively, whereas $VXLAN$ will mask it with $0b11$.

3.1.5 Deparser

We match deparsers emit sequences of header instances with any topological order of header instances extracted from the merged parse graph. We add header instances not extracted during parsing but present in deparsers emit sequences at its topologically ordered location in the graph and regenerate the emit sequence. As P4 generates a list of emitted header instances, we use the merged parse graph to find partial orders among the instances emitted from the deparsers. Also, we verify that every field in every header instance modified in the deparser block of one program is similarly modified by the other program (e.g., checksum computation), otherwise we notify a failure during linking.

3.2 Logical Pipelines Restructuring

The following steps are required for restructuring logical pipelines.

3.2.1 Decomposing CFGs and Constructing Operation Schedules

We apply composition operators by scheduling read-write operations performed on data plane resources by MATs of the programs. First we decompose CFGs and capture schedules of the operations performed on each data plane resources. Each CFG node represents packet processing by a MAT involving match and action phases. We reinterpret the phases as match and action *control nodes* operating on shared data plane resources, as shown in Figure 3a. Match control nodes can only perform read operations on resources representing key fields. Action control nodes can perform read and write operations on resources, because all the actions declared within a table may take resources as input and modify them.

We add stage dependency from match to action control node of the same table. We decompose the CFG shown in 3b using this MAT representation. Figure 3c shows an example where several control nodes in the pipeline access a single resource R1. The decomposed CFG captures all control dependencies of the P4 program among control nodes and their operations on the data plane resources. We create an *operation schedule graph* (called OSG) for each resource from the decomposed CFG to capture all possible sequences of operations that could be performed on the resource at runtime. As a control flow graph (CFG) is a DAG, so are decomposed CFGs and OSGs. Within an OSG, control nodes can be of two types (*node-R, node-W*), depending on the type of

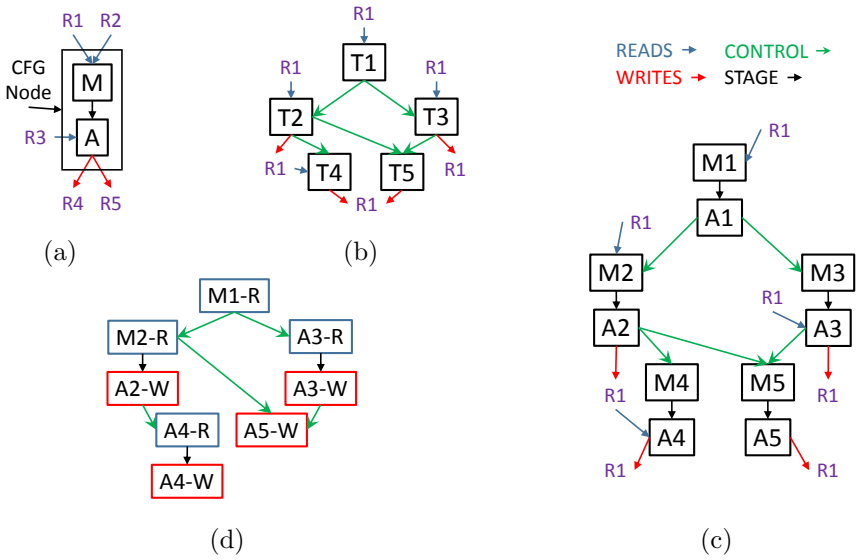


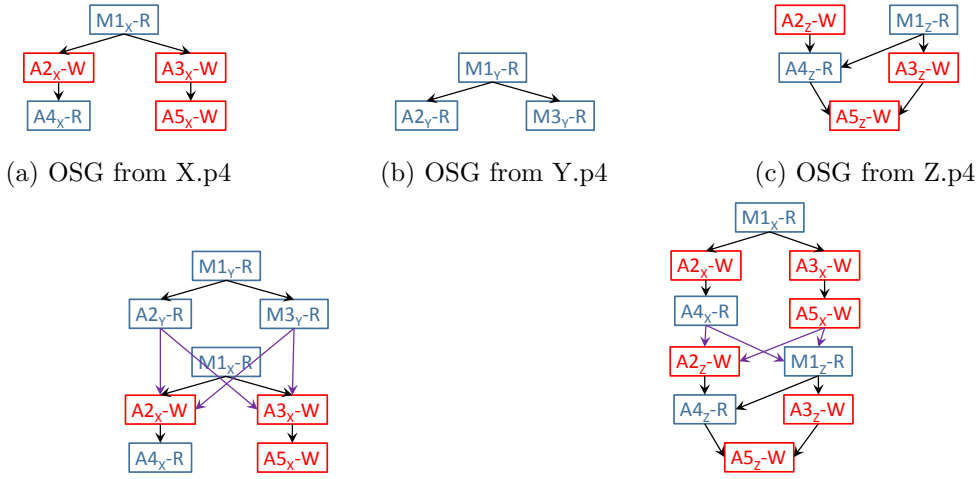
Figure 3: (a) MAT resource dependency; (b) Example of CFG; (c) Decomposed CFG; (d) OSG for resource R1.

operation (i.e., read or write) performed on the resource. Figure 3d shows the OSG for resource R1 created from the graph of Figure 3c. For each data plane resource, we create one OSG per P4 program (derived from its CFG) and we merge all the OSGs by applying the composition operators specified by the network operator.

3.2.2 Applying Composition Operators on Resources

Multiple P4 programs can process traffic in parallel provided that they are processing disjoint data plane resources (or traffic flows). When processing disjoint data plane resources, there can only be one program with write operations on a given data plane resource and all the read operations on this resource from other programs must complete before any possible write operation.

Figure 4a and 4b show an example of OSGs for a resource derived from CFGs of two programs X and Y. All the read operations of the graph from Y must be scheduled before any write operations of the graph from X. To apply parallel composition for shared traffic flows, we add dependencies from all possible last reads of schedule graph from Y to all possible first writes of schedule graph from X, as shown in Figure 4d. In case of programs operating on disjoint traffic flows, we do not add any node dependency across the OSGs and consider merged graph as disconnected acyclic graph. In this case, Runtime enforces isolation of flows of programs in their MATs.



(a) OSG from X.p4 (b) OSG from Y.p4 (c) OSG from Z.p4

(d) Parallel (X ∥ Y) (e) Sequential (X ≫ Z)

Figure 4: Two Different Merging of Resource's OSGs

Once the first program completes all its operations on a data plane resource then and only then the next program is allowed to access the resource. We take an example of merging resource's OSGs from two programs X and Z, shown in Figures 4a and 4c, according to X after Z sequential composition. To apply the sequential composition's constraints, we add dependencies from all possible last operations of X to all possible first operations of Z as shown in Figure 4e. Similarly, we can merge resource's OSGs using any number of P4 programs by adding appropriate node dependencies among them, according to the specified operators and order. We use the merged OSGs of all the data plane resources, while merging CFGs to restructure the logical pipelines.

3.2.3 Merging Decomposed CFGs

We merge decomposed CFGs by creating union of their nodes (i.e., data plane resource and control nodes) and edges. Whatever the type of composition, dependency has to be added between the metadata field that was inserted while merging parsers) and entry tables of the programs. In case of parallel composition, we do not need to add any dependencies across the control nodes, as all the programs should process the assigned traffic flows. However, a merged CFG may have cycles under parallel composition, when for instance program $P1$ writes on a set of resources read by program $P2$ and $P1$ reads a disjoint set of resources written by $P2$. We remove the cycles while recomposing MATs from the merged CFG as described in 3.2.6. In case of sequential composition, some

actions from the first program must be executed before the next program can start processing, even if there is no dependencies between operation schedules of any shared data plane resource from the composition constraint. For example, drop actions from a firewall program must be scheduled before any possible action for the next program. Therefore we add control node dependencies from such specific action control nodes of the first program to all the control nodes without incoming control edge.

3.2.4 Refactoring MATs and mapping to physical pipeline stages

Physical pipeline of a hardware target can be modeled as a DAG of stages with each stage having different memory types with finite capacities and match capabilities, as described in [8]. The DAG provides the execution order of stages and possible parallelism among them. Moreover, decomposed CFGs and merged resources' OSGs provide the scheduling order constraints on 1) match and action control nodes and 2) operations performed by these nodes on the resources. We refactor MATs and orderly map them to the physical pipeline stages by recomposing match and actions control nodes of the merged CFG while respecting the scheduling order and the match stage constraints.

In a OSG, a node-W for a resource without any incoming edge is considered as *ready* to be scheduled. Also, a node-R without any incoming edge along with its exclusive descendants with read operation type are considered ready; e.g., $A2_Z$-W, $M1_Z$-R and $M1_Y$-R, $M1_X$-R, $A2_Y$-R, $M3_Y$-R in Figures 4c and 4d, respectively, are ready nodes. If a control node has no incoming control edge, it is considered as *control-ready*. If a control node has all of its read operations on resources in ready state, it is considered as *complete read-ready*. We select the physical pipeline stages according to their execution order to map the refactored MATs while respecting the constraints of stages. To efficiently utilize the match stage's memory, we relax memory capacity constraints originating from the width of MATs by vertically decomposing them. This allows allocating the match stage memory at the granularity of individual key field instead of all the key fields required to match the control node.

Memory allocation may map either a subset or all of the key fields of a match control node, thereby creating sub matches of the logical MAT. We allocate memory of the current physical match stages by selecting ready node-R pertaining to match control nodes from OSGs for all resources while respecting memory match type and capacity

constraints of physical stages. Once read-ready operations of resources are mapped to the match stage, we remove the nodes and edges from the schedule graphs by making all their predecessors point to their respective successors. For all the sub matches and match control node not having its action node in complete read-ready state, we create sub MATs as described in section 3.2.5 and allocate the actions to the action stage of the physical pipeline. If any complete read-ready match node is mapped to the stage, we schedule its dependent action node provided that it is at least complete read-ready. Also, if any write operation from the action node is not ready, we perform out-of-order write to map the MAT by recomposing match and action control nodes as detailed in 3.2.6.

Parallel composition may create cycles in the merged CFG due to circular dependencies of control nodes from different programs involving more than one resource nodes. Performing out-of-order writes for any resource removes possible cycles involving the resource in merged CFG, originating from parallel composition of programs. Also, it allows to schedule MATs at the earliest stage in the physical pipeline irrespective of existence of cycles in the merged CFG.

3.2.5 Decomposing MATs into sub MATs

We create a new intermediate data plane resource and recompose a MAT with the subset of resources having read-ready operations of a match node to be mapped on multiple nonparallel stages. In this sub MAT, we add a new action setting the intermediate metadata field to the given row ID on successful match at runtime. We consider sub MAT as *secondary table* and intermediate data plane resource as foreign key, which is set by the action in the secondary table. The bit width of the metadata field depends on the length of the MAT defined in its program. Next, we remove dependency edges to the match control node from the selected key field resources. We add the resource dependency of the metadata field to the match control node and add corresponding read-ready operations in the OSG of the field. In successive iterations of mapping physical stages, a sub MAT recomposed with the metadata field resource in its match key fields is considered as the *primary table* being referenced by foreign key of the secondary table. This creates multiple sub MATs using subsets of the match key set, hence vertically decomposing the MAT into sub MATs. We store mappings between

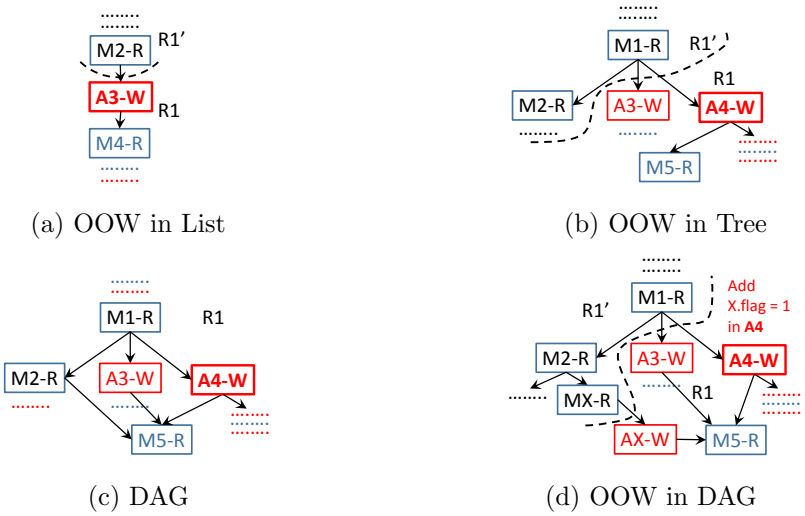
the vertically decomposed MAT and its sub MATs as shown in Figure 1. Also, we use a reverse topological order of stages mapped with sub MATs of vertically decomposed MAT creating an update schedule. The Runtime system uses this schedule to update the sub MATs mapped to the stages of the pipeline.

3.2.6 Out-of-order writing

To explain scheduling of out-of-order write operations in an operation schedule (OS) graph, we start with the simplest case of an OS list, followed by an OS tree and we finally address the general case of an OS DAG in Figure 5. Let us consider the first case shown in Figure 5a, where a resource's operation schedule is a list describing the sequence of operations on it. $A3$-W is scheduled out-of-order by performing all the ancestor operations of the node on the dummy resource R1', hence we split the schedule into two and move the one with all the ancestors to the dummy resource R1'. In case of a tree (shown in Figure 5b with $A4$-W as out-of-order write node), we additionally move all the read operation nodes (e.g., $M2$-R) before any write operation, exclusively reachable from the ancestors (e.g., $M1$-R) of the out-of-order write node. Finally, a tree can be transformed to a DAG by adding edges or paths from a node's siblings or ancestors to its descendants or itself, as shown in Figure 5c. In this case, the descendants of an out-of-order write node reachable using alternative (not involving the write node, e.g., $M5$-R) paths from its ancestors can be scheduled only once all of its other predecessor nodes (e.g., $M2$-R) are scheduled, including the predecessor of the out-of-order write node (i.e., $M1$-R). Also, the value to be used for such nodes ($M5$-R) depends on two resources (R1 and R1') and during execution either of the resource will hold valid value depending on control flow of the execution at runtime. Hence, we add a flag as a data plane resource, a match (MX) and an action node (AX) nodes to copy the value of R1' to R1 by matching on the flag as shown in Figure 5d. This adds one MAT in the merged pipeline in order to recompose and map the MAT to the current pipeline stage using actions node not having all the write operations in ready state.

4 Runtime

The Runtime system aims to manage the tables created by Linker while merging parsers and restructuring the logical pipelines. It allows the control plane of P4 programs to



(a) OOW in List (b) OOW in Tree

(c) DAG (d) OOW in DAG

Figure 5: Out of Order Write Operations (OOWs) and Types of Operation Schedules

insert only the subset of flows assigned by the network operator into the MATs of the pipelines. In order to apply parallel composition operator on disjoint traffic flows, Runtime only needs to enforce traffic flow assignment constraints on programs, because Linker does not recompose MATs of the different programs while restructuring the pipelines. Runtime receives the MAT's flow update from the control plane of P4 programs and uses the header instances UIDT to translate the flow updates from program's header IDs to the Linker IDs. The other major functionality of Runtime is to provide consistency while updating the decomposed MAT, whose sub MATs are mapped to multiple nonparallel stages of the pipeline. Runtime provides consistency for updates in only one MAT of a program, because the table is split by Linker when logical pipelines are restructured.

Runtime maintains referential integrity among the sub MATs of a decomposed MAT. Linker reserves an extra row in sub MATs of a decomposed MAT while mapping its sub MATs to physical pipeline stages. For a flow update in decomposed MAT, if the key fields pertaining to the secondary sub MAT are unique, Runtime first inserts a new match+action entry in the corresponding primary sub MAT using new row ID and rest of the key fields as match. As we use the reverse topological order of stages mapped to sub MATs, Runtime will add a new match entry in the primary sub MATs and then will recursively update the row ID in actions of secondary sub MATs. After insertion in the last sub MAT in update schedule (first in packet processing order), Runtime deletes

old match entries in the sub MATs in reverse order of update schedule, only if it has added new row in the sub MATs.

5 Conclusion

In this report, we outline our first design to compose independently written P4 programs. We describe methods used to merge parser blocks, use ID mappings generated by them in merging of the pipelines. We show that it is possible to merge logical pipelines by considering packet headers and fields as shared resources and by applying composition operations to these shared resources. Deparser is represented as a control block in the P4 language, but we hope that our work provide a good motivation to rethink design of deparser to facilitate conceptually correct merging.

P4Bricks is implemented as an extension of the BMv2 switch simulator. The current implementation includes merging of parsers (3.1.1 to 3.1.3) and restructuring of pipelines (3.2.1 to 3.2.3 & 3.2.5) with out-of-order write operations. An implementation of the selection algorithm to map match key fields to physical stage is ongoing. The complete code with Runtime and Deparser blocks is expected to be released by March, 2018.

Acknowledgements

This work has been partly supported by the French ANR under the "Investments for the Future" Program reference #ANR-11-LABX-0031-01.

References

[1] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[2] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. P5: Policy-driven optimiza-

tion of p4 pipeline. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 136–142, New York, NY, USA, 2017. ACM.

[3] Yu Zhou and Jun Bi. Clickp4: Towards modular programming of p4. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 100–102, New York, NY, USA, 2017. ACM.

[4] David Hancock and Jacobus van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 35–49, New York, NY, USA, 2016. ACM.

[5] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, July 2017.

[6] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Mpvisor: A modular programmable data plane hypervisor. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 179–180, New York, NY, USA, 2017. ACM.

[7] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, pages 13–24, Oct 2013.

[8] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 103–115, Berkeley, CA, USA, 2015. USENIX Association.