An Open-Source P416 Compiler Backend for Reconfigurable Match-Action Table
Switches

Debobroto Das Robin
Kent State University

Javed I. Khan
Kent State University

Abstract

The P4 language has become the dominant choice for programming the reconfigurable match-action table based programmable switches. V1Model architecture is the most widely available realization of this paradigm. The open-source compiler frontend developed by the P4 consortium can execute syntax analysis and derive a hardware-independent representation of a program written using the latest version of P4 (also known as P416). A compiler backend is required to map this intermediate representation to the hardware resources of a V1Model switch. However, there is no open-source compiler backend available to check the realizability of a P416 program over a V1Model switch. Proprietary tools provided by different hardware vendors are available for this purpose. However, they are closed source and do not provide access to the internal mapping mechanisms. Which inhibits experimenting with new mapping algorithms and innovative instruction sets for reconfigurable match-action table architecture. Moreover, the proprietary compiler backends are costly and come with various non-disclosure agreements. These factors pose serious challenges to programmable switch-related research. In this work, we present an open-source P416 compiler backend for the V1Model architecture-based programmable switches. It uses heuristic-based mapping algorithms to map a P416 program over the hardware resources of a V1Model switch. It allows developers to rapidly prototype different mapping algorithms. It also gives various resource usage statistics of a P416 program, enabling comparison among multiple P416 schemes.

1 Introduction

RMT architecture and the P4 language: In recent years, reconfigurable match-action table (RMT) [1] architecture-based programmable switches have become increasingly popular and have seen widespread deployment. The P4 language has emerged as the de-facto standard language to program these switches. Since its introduction [2], the P4 programming language has undergone several architectural changes. Its latest version (version 16 [3], also known as P416)\(^1\) is a major redesign from the initial version (P414 [4]) of the language. It is designed to support various target switches with different architectures (i.e., software switch [5], smartNIC [6], eBPF [7], FPGA [8], RMT [9], dRMT [10] etc.) for packet processing.

\(^1\)Through the rest of the work, by P4, we mean P416 [3], unless mentioned otherwise.

\(\)
programs to be executed by the target hardware. It requires a resource allocation mechanism (Compiler Backend-Mapping Phase in fig. 1) to map the IR components on the target hardware resources. It computes the P4 program’s header field to the RMT hardware’s packet header vector (PHV) mapping, packet header parser state machine (represented as Parse Graph in the IR) to RMT hardware’s state table mapping and the P4 program’s control flow (represented as a graph of logical match-action tables) to RMT hardware’s physical match-action table mapping. These mappings need to be conformant with the target-dependent constraints (i.e., header vector capacity, crossbar width, match-action table dimensions, etc.). If the P4 program can be successfully mapped onto the target hardware; corresponding hardware configurations are generated from the mapping (Compiler Backend-Configuration Generation Phase in fig. 1) in the form of an executable hardware configuration binary. This executable configuration is loaded into the target hardware by the control plane and executed by the target hardware.

Open source P4 compilers for RMT switch: P4C [14] is the reference compiler implementation for the P4 language. It is developed by the P4 Language Consortium and follows [12, 14] the workflow shown in fig. 1. It supports two different RMT architecture based switches: a) simple_switch model widely known as V1Model architecture [15] and b) Portable Switch Architecture (PSA) [16] developed by the P4 Language Consortium (not fully implemented yet). However, P4C does not provide any compiler backend for the real life target hardware of these two architectures. The P4C frontend+midend emits the intermediate representation as a hardware-independent JSON file, and the reference software switch implementation (BMV2 [17]) executes them over a CPU simulated version of the respective hardware architecture. It does not consider the practical hardware resource limitations that exist in real target switches. Hence, P4C can not decide about the realizability of the given P4 program over a specific instance of these RMT switches. Besides the P4C, several other open-source compilers for RMT architecture-based switches are available in the literature. However, some of them [18] work with the older version (P4_14 [4]) of the P4 language, which is architecturally different from the current version of P4 (P4_16 [3]). Some other works focus on different packet processing languages (e.g., Domino [19, 20]), different architecture (dmv [10]), or different hardware platforms (e.g., FPGA [8]). As a result, researchers need to use proprietary compiler backends [21] to decide whether a P4 program can be implemented using an RMT switch or not. However, these systems are closed source, expensive, and often come with additional non-disclosure agreements [9].

Why open-source compiler backend: The compiler backend plays a crucial role in the P4 ecosystem by mapping a P4 program to the target hardware. It is responsible for measuring a P4 program’s resource consumption in the RMT pipeline. Programmable switches contain a limited amount of hardware resources. Therefore, the P4 program with the least hardware resource required to achieve a specific task is more resource-efficient. In recent times, a large number of research works have used the BMV2 [17] simulator with the P4C compiler as their target platform, which lacks a compiler backend that can consider the real-life resource constraints present in the target hardware. Without such a compiler backend, researchers can not measure the resource requirement of their schemes and can not compare the resource usage efficiency of multiple schemes. In the worst case, their P4 program may not be realizable using P4 switches, which are not even identifiable without a compiler backend. Thus, it stands to argue whether these P4 programs are directly executable over real-life RMT switches.

A compiler backend needs to address several computationally intractable problems [18, 22] to find the mapping of a P4 program to the target hardware. The optimal algorithms often require a long time to finish [18, 22]. With the growing rise of the in-network computing [23] paradigm, various research works [24–28] are also focusing on delegating different network functions in the data plane. In these cases, researchers do not need to fit large P4 programs required for full-fledged switches with various features. Optimal mapping algorithms are useful when the data plane programmers need to fit such large P4 programs into target hardware. On the other hand, an open-source compiler backend that uses heuristic-based algorithms can provide the researchers a quick decision about a smaller P4 program’s realizability using a target hardware.

The mapping algorithms used in the compiler backend are sensitive [18] to the resource (TCAM/SRAM storage, number of ALUs, crossbar width, etc.) requirements of a P4 program and the available resources in a target switch. The resource requirements of a P4 program can change at run time (e.g., an increase in the size of an IPv4 forwarding table), which can invalidate a previously computed mapping. With the rapid proliferation of network virtualization [29] and Network-as-a-Service [30] paradigm, the requirement for on-demand network function deployment is also growing rapidly. It requires quick and automated deployment of customized data plane algorithms on a short notice. Therefore, developing faster and more efficient heuristic/approximate mapping algorithms carry enormous significance here. With a closed source compiler backend, researchers can not experiment with different mapping algorithms. Besides this, there is a growing focus [31–33] on developing hardware units for supporting complex instructions (extern [3] in P4 language) in RMT architecture. Without an open-source compiler backend, researchers can not integrate newly developed externs in a P4 program and test their effectiveness. Independently developing compiler backend from scratch requires various common and repetitive tasks (i.e., IR parsing, representing parsed IR using a graph data structure, modeling hardware resources, etc.) not directly related to the computation of the mappings. An open-source compiler backend can allow researchers to focus on developing efficient mapping algorithms rather than focusing on the repetitive tasks.

Inspired by these factors, in this work, we present the design of an open-source P4 compiler backend (mapping phase only) for V1Model [15] architecture-based RMT switches. To the best of our knowledge, it is the first open-source P4_16 compiler backend for RMT architecture based programmable switches. The compiler backend requires two inputs: a) a specification of the available resources in a V1Model switch and b) the intermediate representation (IR) of a P4 program generated by the P4C frontend. As the P4C does not provide any interface to specify the hardware resources of a V1Model switch, we have developed JSON format based hardware specification language (HSL) (sec. 3.3) to express the hardware resource specifications of a V1Model switch. After discussing the related works in sec. 2, we briefly discussed the V1Model architecture in sec. 3 along with the HSL (sec. 3.3). Then we present the structure of the IR provided by the P4C compiler frontend (sec. 4). This backend uses various existing heuristic-based algorithms to allocate resources in the V1Model switch.
pipeline and computes the IR to hardware resource mapping. To the best of our knowledge, this is the first scheme in literature which considers the constraints arising from the use of stateful memory in a P4 program and its impact on the mapping decision. We discussed the details of the mapping process in sec. 5. Once the mapping is found, computing the hardware configuration binaries requires a straightforward translation of the mapping into hardware instruction codes. As this work does not focus on executing a P4 program on any specific instance of V1Model switch, we leave the hardware configuration binary generation for future work. We discuss the implementation and evaluation of our compiler backend in sec. 6 and conclude the paper in sec. 8.

Why V1Model: V1Model is the only RMT switch fully supported by the open-source P4C compiler frontend. Moreover, V1Model architecture is supported by major programmable switch hardware vendors [9, 34]. In recent years, a large set of research works [35] have used the V1Model as their reference hardware architecture (either through the use of commercial hardware or a BMV2 simulator). Moreover, V1Model is similar to the abstract switch model used in the P4 language version 14. Therefore, all the P414 based research works can also be mapped to this model. Finally, the latest programmable switch architecture being standardized by the P4 consortium is PSA [16], and it is also similar to the V1Model architecture. The compiler backend presented in this work can be extended for PSA architecture with a small number of modifications. Due to these reasons, V1Model is a representative hardware architecture for a large number of research works, and we have chosen to build the compiler backend for this architecture.

What the compiler-backend is not: The compiler backend presented in this work supports only V1Model architecture and a subset of P4 language (P416) constructs which cover a wide range of use cases. A full list of P4 constructs supported by the system can be found at [36]. Proprietary hardware can have special instructions (as extern [3]) available for packet processing, and they can also have additional constraints in their system. Our compiler backend is not a full replacement of any proprietary system. It uses heuristic algorithms for mapping a P4 program to V1Model switches, and it can sporadically reject a P4 program (like other programmable switch backends [37]) despite the existence of a valid mapping. Moreover, due to the use of heuristics, it does not guarantee the optimality of the computed mapping. Finally, the compiler backend only covers the mapping phase shown in fig. 1 and does not cover the hardware configuration generation phase.

2 Related Works

In [2], the authors introduced an RMT architecture based abstract switch forwarding model and presented the P4 programming language to program them in a protocol-independent manner. The authors also presented the high-level structure of a two-stage P4 language compiler. Though the work briefly discussed the parser and TDG mapping problem, a complete open-source system for the compiler backend was absent. In [38], the authors addressed the problem of mapping a packet parsing logic to CAM-based hardware. However, its main focus was on synthesizing parser hardware circuitry. Hence it can not be directly used in a P416 compiler backend. In [18], the authors discussed the computational complexity of mapping the logical match-action tables to the physical match-action tables of an RMT switch. They presented an integer linear programming-based method (for optimal solution) and a few heuristic-based methods for computing the mapping. The system is available as an open-source project. However, it can not support stateful memories in the P4 program, which is a crucial requirement for the in-network computation paradigm. All the works mentioned above were designed to support the initial version of the P4 language (a.k.a. P414 [4]), and none of them provides a complete compiler backend. Moreover, the latest version of the P4 language (a.k.a. P416) is architecturally different from the P414. Hence these works can not be used directly for compiling P416 programs.

The reference compiler for the P416 language developed by the P4 language consortium is P4C [14]. Its frontend can compile a P416 program for various target architectures (including the RMT architecture). It provides backend support through CPU-based simulation for two RMT architecture switches: V1Model [15] and PSA [16]. These simulated backends execute the intermediate representation of a P4 program over a CPU. It has no provision to model the hardware resources of an RMT switch. It is unable to consider the hardware resource limit constraints in deciding whether a P416 program can be mapped to target hardware in real life or not. In [8], the authors presented an open-source P416 compiler backend for FPGA-based platforms. However, this system’s basic blocks are different from the physical match-action tables used in RMT architecture. Here, the basic blocks can execute both match, and branching instructions; and based on their result, some actions can be executed. Hence, it provides a more flexible match-action capability in every node compared to the original RMT architecture.

Few other open-source compilers compiler [37, 39] exist in the literature for the RMT architecture. However, they [37] either do not support the program written in P416 as the input or are designed for non-RMT architecture-based hardware platforms [39]. Besides these open-source systems, a few proprietary compiler backends [21, 40] capable of supporting the P416 language for RMT switches also exist. However, they are closed source in nature and do not provide access to their internal mechanisms.

3 V1Model Architecture

The V1Model is an instance of a reconfigurable match-action (RMT) architecture. Its packet processing pipeline (fig. 2) consists of several components arranged in multiple stages. This section describes its components, the specification of different resource types, and how they process a packet. Finally, in sec. 3.3 we present a hardware specification language to represent a V1Model switch’s resources.
3.1 Parser and Packet Header Vector

In V1Model architecture, an incoming packet at first goes through a TCAM based [38] programmable parser (fig. 3), which executes the parsing logic provided in the form of a state machine (converted to a state table by a compiler backend). The parser contains two main building blocks:

a) Header Identification Unit: It contains a $P_B$ bit wide buffer to look ahead in the packet and identify maximum $H$ headers every cycle. It also contains a TCAM capable of storing $P^U_M$ entries to implement the state table. Every TCAM entry contains information of a current parsing state and values (as bit sequence) of header fields to be matched. At every cycle, maximum $f^U_M$ lookup field values (each having maximum lookup width $f^U_W$ b) and the current state can be looked up in the TCAM. The TCAM entries are $P^U_W$ b wide to store the lookup field values and the current state value. Every entry also contains a pointer to RAM cells for storing the next parsing state and the location of the header fields to be extracted by the extraction unit.

b) Extraction Unit: After matching a packet in the TCAM, the information stored at the match index th cell of the SRAM is loaded into the extraction unit. This unit can extract maximum $P^E_M$ bit wide data as header fields and store them in a field buffer. At every cycle, a few header fields are extracted, and the next parsing state is fed to the header identification unit for matching in the TCAM in the next cycle. The header identification unit can move ahead to a maximum $P^U_M$ bit in the packet to start identifying the next header fields. Every parser unit is designed for a maximum parsing rate ($P_{rate}$) throughput. V1Model switches can deploy multiple parser units parallelly for achieving a higher packet parsing rate.

After completing the parsing, all the extracted header fields are sent to a Packet Header Vector (PHV) from the field buffer. The PHV can store $F$ different types of fields; all $f^E_M$ header fields of type $i$ are $f^E_W$ bit wide ($i = 1$ to $F$). Multiple fields in the PHV can be merged together to form a larger header field. Besides the parsed header fields, a PHV also stores hardware-specific metadata (i.e., ingress port, timestamp, etc.). The PHV is passed to subsequent components ($N$ match-action stages fig. 2) in the pipeline through a wide header bus.

3.2 Match-Action Stages

Next, the PHV goes through a series of $N$ match-action stages for ingress stage processing. Each stage (fig. 3) contains $T_M$ units of $T_H$ bit wide TCAM blocks, each capable of storing $T_E$ entries. It also contains $S$ units of $S_W$ bit wide SRAM blocks, each capable of storing $S_E$ entries. The TCAM blocks are used to implement physical match-action tables (MAT) for ternary/range/prefix/exact matching. A fraction of the SRAM blocks ($S^M$ blocks) are used to implement hashtable (using $H_{S^M}$ way Cuckoo hashtable [41, 42]) based physical match-action tables for exact matching, and the rest are used for storing other information (i.e., action arguments, next MAT address, etc.). These smaller physical match-action tables can be run independently or grouped together to match wider header fields within a stage. Physical MATs across the stages can be merged to implement a longer table. Header fields are supplied from PHV to the TCAM and SRAM-based physical MATs through two crossbars, TCB ($TCB_W$ bit wide) and SCB ($SCB_W$ bit wide), respectively. With every entry in the MATs, there is a pointer to corresponding action information (action arguments, action instruction, address of the next MAT to be executed, etc.). On finding a match in the MATs, the corresponding action information is loaded from the memory. Every match-action stage contains a separate arithmetic logic unit (ALU) for every field of the PHV for parallel computation. Two or more units can be grouped together to execute computation on larger fields. Besides the per header field ALU units, a fixed number of extern units (hash, counter, register, meter, etc.) are also available in every match-action stage for special operations (i.e., hash computations, counting, storing/loading states, etc.).

Every stage can store $A_C$ VLIW instructions for all the physical MATs. Every VLIW instruction carries separate instruction for the per header field ALU and extern units. Data is provided to these processing units from PHV through an $ACBW$ bit wide crossbar (ACB). Similar to the match crossbars (TCB and SCB), every bit of this crossbar is driven by all the fields from PHV. The action information (except the action instructions are stored in a dedicated memory) and the stateful memories used by the extern units are allocated in separate chunks of the available SRAM blocks ($S^A$ blocks and $S^S$ SRAM blocks, respectively). Every stage contains $M_p$ memory ports (each one $M_{BW}$ bit wide) capable of read/write from/to an SRAM cell in one clock cycle. These ports are used to read/write data from SRAM blocks for exact MATs, action memory, and stateful memories. Every TCAM-based MAT can store a fixed number of match entries (up to their full capacity). On the other hand, an SRAM-based MAT can store a variable number of entries because the same SRAM blocks are allocated to store match entries, action entries, and stateful memories. The number of total SRAM blocks ($S^M$, $S^A$, and $S^S$ out of total $S$ blocks available) used for exact match MAT, action memory, and stateful memory depends on the logical to physical MAT mapping algorithms of sec. 5.3. To optimize the SRAM usage, RMT architecture allows word packing, creating a packing unit of multiple SRAM blocks. Multiple entries (match, action, or stateful memory entry) can be stored in one unit to reduce SRAM waste. This variable packing format does not impact the match performance, and the match units can match a packet against multiple words stored in the same SRAM block.

Packet Replication Engine and Traffic manager (PRE & TM): After finishing the ingress stage processing, the packet is submitted to the egress port’s queue. The PRE & TM is a non-programmable component responsible for handling the packet’s life cycle in the port’s queues, scheduling the packets, and replicating the packets if necessary. Besides these, there are two more

![Figure 3: A match-action stage in RMT pipeline](image-url)
fixed-function components for computing and verifying the checksum of a packet. As they are fixed-function blocks, we do not discuss their details.

**Egress stage:** Once the packet is picked from the egress port’s queue, it undergoes the egress stage processing. The egress stage is similar to the ingress stage and shares the same physical components for their processing. The compiler backend allocates the resources between the ingress and egress thread in such a way that they do not hamper each other’s packet processing activities.

**Deparser:** After egress stage processing is finished, a packet goes through a deparser block. It recombines the data from the packet header vector fields and the payload. Then the packet is finally propagated through the outgoing channels.

### 3.3 V1Model Hardware Specification Language

A compiler backend requires information about the available resources of a V1Model switch. However, the openly available P4C compiler does not provide any interface to model it. Packet header vector, programmable parser, and match-action stages are the major programmable components in the V1Model architecture. We developed a JSON format-based hardware specification language (HSL) to specify the available resources in the programmable components of a V1Model architecture based switch. The language allows specifying how many header fields can be accommodated in a PHV and what are the bitwidth of these fields. Similarly, it allows specifying the dimension of various hardware resources used in the programmable parser (sec. 3.1). It also allows specifying the number of match-action stages and the number of resources in every stage as described in sec. 3.2. Appendix B shows an example hardware specification of a V1Model switch.

### 4 P4C Intermediate Representation

The P4C frontend takes the **architecture description** of the V1Model architecture and a P4 program as the input. The intermediate representation generated by the P4C frontend (along with the midend) is a **target-independent representation** (IR) of the P4 program. The JSON representation of the IR contains the following major components:

#### 4.1 Header Information

The P4 language provides language constructs to represent a packet header in an object-oriented style where a header object can contain multiple fields. The IR contains a list of all the headers (including the packet metadata header) used in the P4 program along with their member fields and their bitwidth.

#### 4.2 Parse Graph

The **parse-graph** is a directed acyclic graph representation of the parser state machine (parsing logic given in the P4 program). It defines the sequence of headers inside a packet. Every node in the parse graph represents a header type, and the edges represent a transition of the parser state machine. An edge from node $a$ to node $b$ indicates that after parsing header $a$, based on a specific value of one of its member header fields $b$ will be parsed.

### 4.3 Table Dependency Graph

Processing logic for **ingress** and **egress** stages is written using imperative programming constructs given by the P4 programming language. The P4C frontend converts these logics into a flow of logical match-action tables and their dependencies. Thus each stage’s control flow is converted into a **Table Dependency Graph** (TDG). A TDG is a directed acyclic graph where every node represents a logical MAT, and every edge represents the dependency between any two logical MATs. Each node describes the set of fields (header and/or metadata fields) to be matched against the table entries, the types of matching (exact, prefix, ternary, etc.), and the maximum number of entries to be stored in the memory for this table. It also describes the set of actions to be executed based on the match result and the address of the next table to be loaded after executing the current table.

**Non-stateful memory dependency:** Every path in the TDG represents a chain of logical MATs. Following four types of dependencies can arise between any two logical MATs ($A$ comes first and then $B$) in the same path. These dependencies do not involve access to any stateful memory used in the P4 program.

1. **Match dependency:** A field is modified by the actions of $A$, and it is a match field of $B$. Hence execution of $B$’s match part must start after table $A$’s action part has finished execution.

2. **Action dependency:** A field modified by $A$ is not matched in $B$. The same field is also modified by $B$. The modification done by the action of later table $B$ becomes the final result. Hence execution of $B$’s action part must start after table $A$’s action part has finished execution.

3. **Successor dependency:** Whether table $B$ will be executed or not is decided by the match result (hit or miss) of table $A$. Hence execution of $B$’s match part must start after table $A$’s match part has finished execution.

4. **Reverse match dependency:** A match field of table $A$ is modified by the action part of table $B$. Hence execution of $B$’s action part must start after table $A$’s match part has finished execution.

**Stateful memory dependency:** Another type of dependency arises between two logical MATs on the same/different paths if they access the same set of stateful memories (counter, meter, register, etc.). The P4 language provides two different ways for accessing stateful memories: a) **direct:** these stateful memories are attached to a logical MAT and do not create any dependency on other logical MATs. And b) **indirect:** they are not directly attached to any specific MAT, and multiple MAT can access them. Hence they create a dependency among the tables accessing them. The SRAM cells are attached to only one match-action stage in the RMT pipeline. Hence, if a set of SRAM blocks in stage $X$ of the RMT pipelines are allocated for an indirect stateful memory; two or more logical MAT accessing it must need to be mapped on the same match-action stage $X$. We term this as **stateful memory** dependency.

These dependencies influence how the logical MATs in a TDG will be mapped on the physical MATs of an RMT pipeline. The predecessor node in every type of dependency guides the starting clock cycle of its successor [1, 18]. Hence, they also determine the total processing delay of a packet undergoing the processing defined by a P4 program. However, the TDG in the IR itself has no mechanism to represent various types of dependencies directly. A compiler backend needs to analyze the TDG to identify...
the dependency between two logical MAT before computing the mapping.

5 IR to Hardware Resource Mapping

The goal of the mapping phase shown in Fig 1 is to map a given P4 program to the resources of a V1Model switch. A compiler backend takes two types of information as input for this purpose: (a) the hardware resource specification of the switch described using the HSL presented in Sec. 3.3 and (b) the hardware-independent intermediate representation (IR) (Sec. 4) of the given P4 program generated by the P4C compiler frontend. To determine whether the given P4 program is realizable over a V1Model switch, the compiler backend needs to address three different types of mapping problems: (a) header mapping: mapping the IR’s header fields to the PHV fields b) parse graph mapping: mapping the parse graph to parser hardware resources, and c) TDG mapping: mapping the TDG to the resources of match-action stages. While addressing these problems, the compiler backend needs to ensure that a computed mapping does not violate the constraints (both architectural and resource constraints) of the target hardware and the control flow of the P4 program. Besides computing a valid mapping, the backend also needs to maximize the concurrency and resource usage efficiency of the P4 program.

Computing an optimal and valid mapping of a P4 program to a V1Model switch is a computationally intractable problem [18, 22]. Our compiler backend uses heuristic-based algorithms to find a mapping while maintaining the architectural and resource constraints of an RMT switch. We describe how our compiler backend works to find these three types of mapping through the rest of this section using a simple QoS-modifier program (see Appendix A for the P4 source code). In this program, the control plane supplies a separate QoS value for IPv4 and IPv6 packets forwarded through an individual port. The P4 program stores these values in two indirect stateful memories (register array) on receiving the control packet from the control plane (defined in MAT named match_control_packet). For every valid IPv4 packet, if it matches with the entries configured in ipv4_next-hop, that packet’s diffserv value is replaced with a value read from the register array (ipv4_port_qos), and its IPv6 destination is set as a server’s IP address. Then the packet is matched with ipv6_nexthop MAT to set its IPv6 trafficClass after reading from another register array (ipv6_port_qos). In the example, we have used the benchmark V1Model switch specified in Appendix B.

5.1 Header Mapping

V1Model switches contain a limited number of fixed bitwidth fields in the PHV (provided as a part of the hardware specification). The header fields used in the program is needed to be accommodated using these PHV fields to execute a P4 program. For example, a 17b wide header field can be accommodated using three 8b or 32b wide PHV fields. In the first case, the amount of resource waste is 7b and 15b in the latter case. Here, the mapping algorithm also needs to minimize the amount of waste. Besides this, as the V1Model switches contain two different processing threads (for the ingress and egress stage) sharing the same hardware pipeline, every PHV field needs to be explicitly allocated to one of them.

| P4 Program Header Fields | PHV Fields |
|--------------------------|-----------|
| Bitwidth | Count | Bitwidth | Count | Bitwidth | Count |
| 1 | 13 | 1 | 16 | 6 |
| 3 | 19 | 2 |
| 4 | 20 | 1 |
| 7 | 32 | 7 |
| 8 | 48 | 2 |
| 9 | 128 | 2 |

Table 1: Bitwidth of QoS-Modifier program’s (Appendix A) header fields and bitwidth of PHV fields required to accommodate them.

The problem of optimally allocating the PHV fields (set of items) to accommodate the P4 program’s header fields (set of resources) can be modeled as a multiple knapsack problem [43]. In this case, we need to find how to optimally fill up a set of bins (the header fields in the P4 program) using the set of items (all the PHV fields) while minimizing the waste. As the problem is computationally intractable, we used a heuristic-based (fill the largest header field at first using the largest bitwidth PHV field available) algorithm to compute the mapping. The algorithm first analyzes the header information in the IR (Sec 4.1) and makes two disjoint sets of header fields used in the ingress and egress stage. The metadata fields are needed to be replicated for both stages. The header fields are sorted in descending order of their bitwidth. For each header field in sorted order, the algorithm picks the largest bitwidth PHV field from the remaining PHV field that leads to the least amount of waste. The process continues until the bitwidth of a header field is filled with selected PHV fields. Table 1 shows the bitwidth of the header fields used in the QoS-Modifier program (Appendix A). It also shows the bitwidth of the PHV fields, and how many of them are required to accommodate those header fields according to the mappings computed by the compiler backend.

5.2 Parse Graph Mapping

The IR gives a graph (parse graph in Sec. 4.2) representation of the parser state machine. On the other hand, the parser in the V1Model switches follows match-action semantics. In every cycle, the parser hardware can look into the packet header, identify a limited number of header fields, and match them with the header-identification unit’s TCAM entries (works as a state table). On finding a match in the TCAM, the corresponding action (header fields extraction by the extraction-unit) is executed. The compiler backend needs to convert the parse graph into a state table which needs to be accommodatable within the dimensions of the parser TCAM. Similarly, the field extraction operations must be accommodatable within the capacities of the extraction unit.

Generating an optimal state table from a parse graph and relevant TCAM entries is a computationally intractable problem [38]. Our compiler backend uses the algorithm presented in [38]. This algorithm tries to find the clusters in the parse-graph which can be accommodated within the capacity (lookup in the packet header and extracting header fields) of the parser hardware. For every unique pair of a cluster and an edge (transition in the parser state machine) in the cluster graph, one entry is required in the TCAM based parse-table. The parser hardware executes the match-action
logic for every cluster in one cycle and moves to the next parsing state. We converted the source code of [38] to work with the IR generated by P4C and used it in the compiler backend to generate the state-table entries. If the entries cannot be accommodated using the TCAM, then the P4 program is rejected. As an example, the compiler backend was used to compute the parse-graph mapping for the P4 program shown in appendix A over the V1Model hardware described in appendix B. Fig. 4 shows the corresponding parse graph, identified clusters in it, and the corresponding TCAM entries.

5.3 TDG Mapping

The TDG expresses the packet processing logic of a P4 program as a control flow among logical match-action tables guided by their dependencies. The compiler backend needs to map these logical MATs over the physical MATs of the V1Model switch while preserving the original control flow. The computed mapping must not violate the following constraints.

- The match width of all the logical MATs mapped to any match-action stage must not exceed the crossbar width for TCAM based MATs (TCBW) and SRAM based MATs (SCBW).
- The bitwidth of the match fields of every logical MAT must not exceed the total match width of the mapped physical MATs.
- The total number of entries required for any logical MAT must not exceed the capacity of the mapped physical MATs.
- The total width of the fields used as the action inputs must not exceed the width of the action crossbar of a match-action stage.
- The total number of actions (using ALU or extern units) executed on an individual PHV field in a match-action stage must not exceed the available number of actions for that stage.
- The total amount of SRAM required by logical MATs for action memories and stateful memories (using a register, meter, counter, etc.) must not exceed the available SRAM volume in a match action stage.
- If two or more logical MAT nodes access the same indirect stateful memory they need to be mapped on the same physical match-action stage.
- The physical match action tables in a single stage can be executed concurrently. However, due to the dependencies (sec. 4.3), the compiler backend needs to map the logical MATs over the physical MATs in such a way that does not violate the dependencies.

Besides ensuring the constraints, the compiler backend needs to minimize the number of match-action stages and amount of resource usage in each of the used match-action stages. The first one leads to reduced processing delay of a packet [18]. The second one leads to an overall reduction in resource usage and power consumption by a V1Model switch. Finding a mapping that has optimal resource usage and does not violate the upper mentioned constraints is a computationally intractable problem [18, 22].

5.3.1 Our approach

The TDG mapping problem can be modeled as an integer linear programming [18] problem, and an optimal mapping can be determined (if it exists). However, it may take a large amount of time [18, 22] to compute such a mapping. Hence we applied heuristic-based algorithms to compute the mapping. The algorithm works as follows: First, the algorithm preprocesses (sec. 5.3.1) the TDG to transform the conditional nodes into MAT. Then it loads the TDG into a graph-based data structure with dependencies among the nodes as edge. It checks whether the stateful memory dependencies in the TDG can be met on the target hardware or not. It also transforms the logical MATs to meet the target hardware constraints if necessary. Finally, the algorithm maps the logical MATs to the physical match-action stages.

If a dependency chain in the TDG contains $n$ logical MATs, it requires at least as many match-action stages in the V1Model pipeline. Similar to [18], we define the level of a logical MAT as the number of dependencies remaining in any dependency chain starting from that node. Assigning the levels requires a topological ordering of the logical MATs in the TDG. Due to topological ordering, all logical MATs at the same level have no dependency on each other. Then the mapping algorithm tries to map all the logical MATs of the same level. This process is repeated for all levels.

**TDG preprocessing** The TDG generated by the P4C frontend transforms every conditional branching instruction (if/else pair) in the P4 program to three logical MATs. Fig. 5a shows the TDG of the P4 code presented in appendix A. The topmost node shows the TDG node for evaluating the conditional expression, and the next two nodes show its two branches (one each for true and false) as the next nodes. However, the task for evaluating the conditional expression is executed in the action portion of the first MAT, and it does not pass the result of the expression evaluation to the next nodes. Hence, a packet can successfully match with both the next nodes, which is incorrect. To avoid this, we modify the conditional evaluation expression step with a conditional assignment available [1,9] in V1Model switches. It assigns 1 to an auxiliary header
Stateful Memory Dependency can exist between logical MATs in the same or different paths in the TDG. Indirect stateful memories accessed by two (or more) logical MATs in the same path implies non-stateful memory dependency (sec. 4.3) exists among the logical MATs. Hence they need to be mapped on different physical match-action stages. However, they need to access the same stateful memory and the RMT architecture does not allow access to the same SRAM block from different match-action stages. Therefore, these kinds of P4 program can not be implemented using V1Model switches (though the P4 language syntax allows it), and our algorithm rejects them.

On the other hand, logical MATs on different paths in the TDG with stateful memory dependency need to be mapped on the same physical match-action stages. Hence they need to be assigned the same level. Besides this, a logical MAT may need to be divided into two separate logical MATs for valid mapping. For example, consider the case of three logical MATs ipv4_nexthop, ipv6_nexthop, and match_control_packet in the TDG of fig. 5a. Here match_control_packet is writing over two stateful memories ipv4_port_qos and ipv6_port_qos; ipv4_nexthop is reading from only ipv4_port_qos and ipv6_nexthop is reading from only ipv6_port_qos. Now, there exists a match-depending between ipv4_nexthop and ipv6_nexthop; hence they can not be mapped to the same stage. Therefore, the stateful memories accessed by these logical MATs also can not be mapped to the same stage. Assume ipv4_port_qos and ipv6_port_qos are mapped to stage X and Y in the pipeline; therefore, ipv4_nexthop and ipv6_nexthop are needed to be mapped on stage X and Y, respectively. However, this requires match_control_packet to be mapped on both stage X and Y; clearly this is not possible. To accommodate such scenarios, we bifurcate match_control_packet’s action into two half based on the stateful memory access (fig. 5c). The first half contains the original match and action part up to accessing ipv4_port_qos, and another logical MAT (match_control_packet–Part-2) is created without any match field and only the remaining part of the match_control_packet’s actions (accessing ipv6_port_qos). This new logical MAT is added between match_control_packet and its successor in the TDG.

Level Generation In the TDG, there can be multiple dependencies between logical MATs in the same path; however, the strictest dependency (among the four types of non-stateful memory dependencies) is the main factor affecting the mapping decision [18]. The mapping algorithm keeps the strictest dependency and removes other dependencies. Besides this, all the logical MAT nodes having stateful memory dependency are needed to be assigned the same level to ensure they are mapped on the same physical match-action stage. After the preprocessing step, the TDG nodes are topologically ordered to label them with their appropriate levels. This ordering ensures that every node with any non-stateful memory dependency is assigned a higher level than its successors. Finally, if two neighbor nodes in the TDG only have successors or reverse match dependencies or no dependency, they can be mapped to the same physical match-action actions (as they can be executed concurrently or speculatively) and assigned the same level. Fig. 6 shows the preprocessed version of the TDG shown in fig. 5, along with the levels assigned to every node.

Mapping Logical Tables After level generation, all the logical MATs with the same level imply they can be mapped to the same physical match-action actions. They can be divided into two subsets. Firstly, a subset of these logical MATs contain stateful memory dependency among them, and they need to be mapped on the same stage where the indirect stateful memories are mapped. Similar to existing commodity RMT switches [9], our compiler backend does not support spreading indirect stateful memories over multiple stages. The mapping algorithm at first maps the set of logical MATs containing stateful memory dependency to only one of the available physical match-action actions. Secondly, the following subset of logical MATs contains no match, action, or stateful dependencies, and these logical MATs can be executed concurrently on the same match-action stage. However, a single match-action stage may not contain enough hardware resources to accommodate them. In that case, they are mapped to one or multiple consecutive match-action stages.

Logical MATs from these two sets can be ordered and selected for mapping to the physical match-action stage using various heuristics [18]. In this work, we prioritized the logical MATs with non-exact (ternary, lpm, or prefix) match fields and mapped them to TCAM-based physical match-action blocks. Next, the logical MATs with exact match fields are mapped to SRAM-based physical match-action units. These logical MATs can spill into TCAMs if it runs out of SRAMS. Ties are broken by at first mapping the logical MAT that appears in the TDG earlier. Fig. 6 shows the logical MAT nodes of the TDG shown in fig. 5 and on which physical match-action stages they are mapped.

In allocating SRAM blocks for the match, action, and stateful
memory entries, the mapping algorithm utilizes the memory packing feature of RMT architecture. The mapping algorithm tries to store multiple entries in a packing unit of up to \( p_f \) SRAM blocks to reduce SRAM waste and fragmentation. The value of \( p_f \) is configurable at compile time. Currently, our compiler backend allocates SRAM at block level granularity. As a result, an SRAM block is allocated exclusively for the match or action entries of only one MAT; or for accommodating the indirect stateful memories. It can lead to a waste of SRAM resources when a small number of entries are required. We leave the goal of improving the SRAM utilization as future work. The number of action entries required for a logical MAT can be determined at compile time [25] or unknown beforehand [18]. Our compiler backend can reserve either a fixed number of action entries or one action entry for every MAT entry in a logical MAT.

6 Implementation and Evaluation

6.1 Implementation

Our P4 compiler backend is entirely implemented in the Python 3 programming language. For the frontend, it relies on the P4 consortium’s reference compiler implementation (P4C [14]), which parses the P4 source code and generates an intermediate representation in JSON format. Our backend parses this JSON format data and stores it in a graph-based data structure. For computing the parse graph to state table representation for the parser TCAM, we relied on the algorithms proposed in [38]. This project’s source code was written in Python 2 language, and it was not designed for P4_16’s intermediate representation of P4C. We converted the source code in Python 3 language and integrated it with our project with a moderate level of modifications. All source code of the project is publicly available [36] under an open-source license.

6.2 Evaluation

In this section, we analyze the performance of the proposed P4 compiler backend presented in this work. Due to various factors, evaluating and comparing its performance with other compiler backends is challenging. A few of the important reasons are the following: a) Due to the lack of openly available complete compiler backend (which can compute all three types of mapping for a P4 program) for RMT switches, we cannot compare the overall performance of our compiler backend. b) To the best of our knowledge, there are no benchmark p4 programs and complete information about their resource consumption in the RMT hardware pipeline is available. Few research works have mentioned corresponding P4 program’s resource consumption using proprietary compilers. However, these target hardware often contains various externs capable of executing complex actions. These externs can significantly impact the mapping of a P4 program. Hence, it is not appropriate to use them for comparison without knowing the details of the target hardware and relevant mapping algorithm. Moreover, a large number of research works are based on the older version of the P4 language (P4_14). P4C can compile both P4_14 and various P4_16 programs targeted for V1Model switches. However, we have found several instances [26, 44] of P4_14 programs used in literature are not compilable using P4C due to a lack of proper backward compatibility. Besides this, we have found that several P4_16 programs [26, 45] are also not compilable with P4C due to changes in some of the APIs in BMV2 [17] implementation. To this end, we have selected the following four P4 programs to evaluate our compiler backend’s performance.

a) IPv4/IPv6 QoS modifier: This program is discussed in 5 and the P4 source code is presented in appendix A.

b) Simple layer-2/3 forwarding: This P4 program is designed for simple layer-2/3 packet forwarding and written using P4 version 14. It is compatible to P4_16 and used in the TDG mapper proposed in [18].

c) Complex layer-2/3 forwarding: This P4 program is a complex version of the previous one and requires more resources. It is also written using P4 version 14. It is compatible to P4_16 and used in the TDG mapper proposed in [18] also. Both versions of the layer 2/3 forwarding program was adopted after minor modification to make it compatible with P4C. Moreover, TDG mapping of this P4 program is mainly influenced by the availability of memory (SRAM and TCAM) is the pipeline stages. Hence, it can be considered as a memory intensive P4 program.

d) Traffic Aanonymizer: This program is an implementation of the scheme proposed in [46] to anonymize a packet’s content. The program is fully realizable using Tofino [47] switches and BMV2 based source code is available as open-source project. This program has a small stateful memory (SRAM and TCAM) requirement; however, it has a complex TDG and mainly requires computational power in various stages. Hence, it can be considered as a compute intensive P4 program.

For each P4 program, we generated the intermediate representation (IR) using the P4C frontend. We used the V1Model switch described in appendix B as the benchmark hardware. We provided its hardware specification and the IR to the compiler backend as input. Then we used our compiler backend to map the IR of the P4 programs to this benchmark hardware. As there is no open-source compiler available for computing the header and parser mapping of a P4 program, we have only presented the result using our compiler backend. However, for TDG mapping, we have computed the mapping using the TDG mapper presented in [18] (with First-Fit-Decreasing heuristic) and compared it with the mapping generated by our backend. The TDG mapper’s [18] mapping algorithm reserves a fixed number of SRAM blocks (16 blocks per stage [18]) for accommodating action memories and can not dynamically adjust the number of action entries (for one or more logical MATs) beyond the limit of 16 SRAM blocks. As it reserves a fixed number of SRAM blocks, it does not require to compute the mappings of the action entries over the physical MAT stages. Besides the heuristic based mapping, this TDG mapper [18] can produce optimal mapping of the logical MATs to physical MATs; However, for optimal mapping it requires a large amount of time [18, 22] to compute the mapping using integer linear programming (ILP) method. As our goal is to build a compiler backend for to quickly decide about realizability of a P4 program, we have not included the ILP variation of the TDG mapper proposed in [18]. Compared to the TDG mapper of [18], we have configured our compiler backend to allocate up to 16K action entries for every logical MAT in every stage. All of our experiments were run on an HP laptop with an Intel Core-i7 processor, 24 GB RAM, running Ubuntu 20.04.
6.2.1 Result Analysis

Header Mapping: Table 2 shows the result of our compiler backend’s header mapping of the benchmark P4 programs. There are 64, 96, and 64 fields of 8, 16, and 32b width (total 4096b wide PHV) exists in the PHV. Many PHV fields remain unused for both small and large programs; for example, the small programs (QoS-Modifier and L2L3-Simple) both consume around 30-35% of the PHV’s capacity. On the other hand, the large programs (Traffic-Anony and L2L3-Complex) consume around 51% of the PHV’s capacity. While accommodating the P4 program’s header fields using the PHV fields, some space in the PHV fields is wasted. For the small programs, the waste in PHV bitwidth is higher (approx. 10-12%) compared to the larger programs (approx. 5.7-6.4%).

Parse Graph Mapping: Table 3 shows the result of our compiler backend’s parse graph mapping of the benchmark P4 programs. The benchmark hardware used in the experiments contains 256×40b TCAM for implementing the parser state table; it can look at 48 bytes into the packet, identify a maximum of four headers and extract 48 bytes of header fields data at every cycle to parse the incoming packets at 40 Gbps. For the complex P4 program (L2L3-Complex) with 11 states and 31 edges in the parse graph, it requires only 22 entries in the TCAM, which is less than 9% of the total capacity of the TCAM. The parse graph is simpler for the rest of the programs and consumes only 2% of the TCAM’s capacity. The rightmost column of table 2 shows the total time (in milliseconds) required to compute the header mapping for the benchmark P4 programs. For all the P4 programs, the required time is short and ranges between 2 to 3 milliseconds (approx.).

TDG Mapping: Table 4 shows the comparison between the TDG mapping computed by the TDG mapper of [18] and our compiler backend. For the QoS-Modifier program, the TDG mapper proposed in [18] computes the mapping wrongly. It maps match_control_packet and ipv4_nexthop in one stage and ipv6_nexthop in another stage. This is an invalid mapping, as explained in sec. 5.3.1 and fig. 5. The reason behind the invalid mapping (row 1 in table 4) is that the TDG mapper proposed in [18] only focuses on the four types of non-stateful memory dependencies (sec. 4.3) in computing the mapping. However, the QoS-Modifier program contains a sequence of logical MATs where both match and stateful memory dependency guide the control flow. Hence, the mapper fails to compute a valid mapping for the QoS-Modifier program. On the other hand, our compiler backend includes stateful memory dependency and the other four types of non-stateful memory dependencies in computing the TDG mapping. It generates a valid mapping for this P4 program. It maps the logical MATs over three physical match-action stages, and the processing latency of every packet under this mapping is 38 cycles. It requires 6 TCAM blocks and 4 SRAM blocks to accommodate the logical match-action tables.

The Traffic-Anony program is written in P4 16. Whereas the work described in [18] does not support various P4 16 language constructs and only works with the P4 14 programs. Hence it cannot generate the mapping for the Traffic-Anony P4 program (row 2 in table 4). However, our compiler backend supports the P4 16 language and generates a valid mapping for the program. It maps the P4 program over 18 physical match-action stages and the processing latency of every packet under this mapping is 156 cycles. The Traffic-Anony is a computation-intensive P4 program; it requires 2 TCAM blocks and 35 SRAM blocks to accommodate the logical match-action tables in the RMT pipeline.

The P4 programs for simple and complex L2L3 forwarding (L2L3-Simple and L2L3-Complex) are both written in P4 14. None of the programs contain any stateful memory dependency. Hence their TDG mapping can be computed using the work proposed in [18]. In the case of L2L3-Simple (row 3 in table 4), our compiler backend uses five stages but achieves reduced packet processing latency of 42 cycles. On the other hand, the TDG mapper of [18] uses one less stage, but the packets face more processing latency (53 cycles). Our compiler backend requires one extra TCAM block and 16 SRAM blocks to accommodate the logical MATs. In the case of L2L3-Complex, our compiler backend uses 31 stages which is one stage more compared to [18]. However, it needs a smaller packet processing latency of 108 cycles. Our compiler backend uses 260 TCAM blocks compared to 272 TCAM blocks used by the TDG mapper of [18]. However, our compiler backend uses a higher number of SRAM blocks (995 blocks compared to 762 blocks). Overall our compiler backend requires more SRAM blocks for accommodating the simple and complex L2L3 forwarding programs. There are two reasons behind this: Firstly, our compiler backend allocates SRAM blocks for action memory and indirect stateful memories at a granularity of blocks. As a result, when the SRAM requirement of two or more logical MATs can be fulfilled using only one SRAM block, our compiler backend allocates at least one SRAM block to every logical MAT. This memory packing is less efficient compared to [18]. We leave the goal of improving the SRAM allocation mechanism to reduce such waste as a future goal. Finally, our compiler backend does not reserve a fixed number of SRAM blocks for action memories.
in every stage. Instead, it can store up to 16K (can be increased or
decreased at compile time) action entries for every logical MAT.
It sacrifices cheaper SRAM uses for more flexibility in action
memory allocation. As a result, it tends to use more SRAM blocks
compared to the TDG mapper of [18].

**Execution time:** The TDG mapper of [18] can not compute a
valid TDG mapping for the QoS-Modifier and Traffic-Anony pro-
gram; hence execution time for these two benchmark P4 programs
can not be compared. However, in the case of L2L3-Complex, our
compiler backend requires approximately 6x time compared to the
TDG mapper of [18] (column 12-13, last/5-th row in table 4)
to compute the TDG mapping. There are two major reasons be-
hind this: Firstly, our compiler backend preprocesses (sec. 5.3.1)
the TDG to handle the stateful memory dependency, which in-
creases the execution time. Secondly, the TDG mapper’s [18]
policy of allocating a fixed number of SRAM blocks per stage for
action memories does not require any computation at run time.
Conversely, our compiler backend dynamically allocates action
memories for every logical table and also tries to minimize the
usage of SRAM blocks. Hence it needs to search over a larger
search space which results in a larger execution time. The L2L3-
Complex is a memory-intensive P4 program; hence execution time for these two benchmark P4 programs
can not be compared. However, in the case of L2L3-Complex, our
compiler backend requires approximately 6x time compared to the
TDG mapper of [18] (column 12-13, last/5-th row in table 4)
to compute the TDG mapping. There are two major reasons be-
hind this: Firstly, our compiler backend preprocesses (sec. 5.3.1)
the TDG to handle the stateful memory dependency, which in-
creases the execution time. Secondly, the TDG mapper’s [18]
policy of allocating a fixed number of SRAM blocks per stage for
action memories does not require any computation at run time.
Conversely, our compiler backend dynamically allocates action
memories for every logical table and also tries to minimize the
usage of SRAM blocks. Hence it needs to search over a larger
search space which results in a larger execution time. The L2L3-
Complex is a memory-intensive P4 program; hence execution time for these two benchmark P4 programs
can not be compared. However, in the case of L2L3-Complex, our
compiler backend requires approximately 6x time compared to the
TDG mapper of [18] (column 12-13, last/5-th row in table 4)
to compute the TDG mapping. There are two major reasons be-
hind this: Firstly, our compiler backend preprocesses (sec. 5.3.1)
the TDG to handle the stateful memory dependency, which in-
creases the execution time. Secondly, the TDG mapper’s [18]
policy of allocating a fixed number of SRAM blocks per stage for
action memories does not require any computation at run time.
Conversely, our compiler backend dynamically allocates action
memories for every logical table and also tries to minimize the
usage of SRAM blocks. Hence it needs to search over a larger
search space which results in a larger execution time. The L2L3-

Our compiler backend’s larger execution time can be observed
in the case of the Traffic-Anony program also. It is a computa-
tion-intensive P4 program (3-rd row in table 4) containing a large
number of nodes and edges in the TDG (84 nodes and 194 edges)
due to the nested branching instructions. However, it requires very
little memory (only 2 TCAM and 35 SRAM blocks). Hence, the
TDG mapping of this program is mainly influenced by the available

| Program Name   | # Nodes in TDG | # Edges in TDG | Stages | Latency (in cycle) | Resource Usage (in Blocks) | Ex. Time (in ms) |
|----------------|----------------|---------------|--------|-------------------|---------------------------|-----------------|
|                |                |               |        |                   | TCAM | SRAM |          |                  |                  |
| QoS-Modifier   | 16             | 20            | Inv. 3 | Inv. 38           | Inv. 6 | Inv. 4 | Inv. 31 |                  |                  |
| Traffic-anony  | 84             | 194           | Inv. 18| Inv. 156          | Inv. 2 | Inv. 35 | Inv. 1700|                  |                  |
| L2L3-simple    | 24             | 38            | 4      | 53                | 56    | 191    | 207     | 243              | 141              |
| L2L3-Complex   | 60             | 138           | 30     | 310               | 272   | 762    | 995     | 148              | 925              |

Table 4: Comparison of TDG mapping computed by the work proposed in [18] and the compiler backend presented in this work (marked by *): Inv. = Invalid mapping.

7 Discussion

**Limitations:** Our compiler backend supports most of the P4 lan-
guage constructs covering a wide range of use cases. However,
it still does not support variable-length header parsing and direct
stateful memory access in actions. Both of them can be avoided
through careful design of the p4 program. Besides this, it does
not support the atomic transaction mechanism available in the
P4 language. We are working on supporting these P4 language
features.
We have presented an open-source compiler backend that can map a P4 (version 16) program to the hardware resources of a V1Model switch. It uses heuristic-based algorithms to compute different mapping algorithms and different variations of V1Model architectures supporting various extern units. This can provide an open platform for analyzing the realizability and resource consumption of a P4 (version 16) program in real-world V1Model switches. It allows researchers to experiment with different mapping algorithms as an open-source platform. It can be extended to support other derivatives of V1Model architecture by supporting various extern units. This can provide an open platform to the programmable switch researchers for experimenting with different mapping algorithms and different variations of V1Model switch.

8 Conclusion

We have presented an open-source compiler backend that can map a P4 (version 16) program to the hardware resources of a V1Model switch. It uses heuristic-based algorithms to compute this mapping and give a quick decision on the realizability of a P4 program. We believe this open-source compiler backend can serve as a cost-effective platform for analyzing the realizability and resource consumption of a P4 (version 16) program in real-world V1Model switches. It allows researchers to experiment with different mapping algorithms as an open-source platform. It can be extended to support other derivatives of V1Model architecture by supporting various extern units. This can provide an open platform to the programmable switch researchers for experimenting with different mapping algorithms and different variations of V1Model switch.

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. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[3] The P4 Language Consortium. P416 language specification. https://p4.org/p4-spec/docs/P4-16-v1.2.1.html.

[4] The P4 Language Consortium. P414 language specification. https://p4.org/p4-spec/p4-14/v1.1.0/tebx/p4.pdf.

[5] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 525–538, 2016.

[6] Yonatan Piasetzky, Matty Kadosh, Marian Pritsker, Omer Shabtai, Alan Lo, and Guohan Lu. Switch asic programmability in hybrid mode. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 448–449. IEEE, 2018.

[7] William Tu, Fabian Ruffy, and Mihai Budiu. Linux network programming with p4. In *Linux Plumb. Conf.*, 2018.

[8] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, pages 122–135, 2017.

[9] P416 intel® tofino™ native architecture – public version application note. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf, 2021.

[10] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dmrtn: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 1–14, 2017.

[11] Debobroto Das Robin, Dr Khan, and I Javed. Toward an abstract model of programmable data plane devices. *arXiv preprint arXiv:2008.08697*, 2020.

[12] Mihai Budiu and Chris Dodd. The architecture of the p416 compiler, 2016.

[13] Tharun Kumar Dangeti, Ramakrishna Upadrashta, et al. P4llvm: An llvm based p4 compiler. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 424–429. IEEE, 2018.

[14] The P4 Language Consortium. P416 reference compiler - github. https://github.com/p4lang/p4c.

[15] The P4 Language Consortium. v1model.p4 - architecture for simple_switch. https://github.com/p4lang/p4c/blob/master/v1model.p4.

[16] The P4 Language Consortium. P416 portable switch architecture (psa). https://p4.org/p4-spec/docs/PSA-v1.1.0.pdf.

[17] The P4 Language Consortium. Behavioral model (bmv2). https://github.com/p4lang/behavioral-model. Accessed: 2022-04-10.

[18] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115, 2015.
[19] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet translations: High-level programming for line-rate switches. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 15–28, 2016.

[20] Xiangyu Gao, Taeyyun Kim, Michael D Wong, Divya Raghu-nathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 44–61, 2020.

[21] Intel Corporation. Intel® p4 studio. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html.

[22] Balázs Vass, Erika Bérczi-Kovács, Costin Raiciu, and Gábor Rétvári. Compiling packet programs to reconfigurable switches: Theory and algorithms. In Proceedings of the 3rd P4 Workshop in Europe, pages 28–35, 2020.

[23] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In Proceedings of the 16th ACM Workshop on Hot Topics in Networks, pages 150–156, 2017.

[24] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay Thakur, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A p4-based 5g user plane function. In Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR), pages 162–168, 2021.

[25] Debobroto Das Robin and Javed I Khan. CLB: Coarse-Grained Precision Traffic-Aware Weighted Cost Multipath Load Balancing on PISA. IEEE Transactions on Network and Service Management, 2022.

[26] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. ACM SIGCOMM Computer Communication Review, 46(2):18–24, 2016.

[27] Debobroto Das Robin and Javed I Khan. P4TE: PISA switch based traffic engineering in fat-tree data center networks. Computer Networks, page 109210, 2022.

[28] Debobroto Das Robin and Javed I Khan. P4KP: QoS-Aware Top-K Best Path Using Programmable Switch. IEEE Access, 9:109115–109129, 2021.

[29] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In Proceedings of the 12th International Conference on emerging Networking EXperiments and Technologies, pages 35–49, 2016.

[30] Minqi Zhou, Rong Zhang, Dadan Zeng, and Weining Qian. Services in the cloud computing era: A survey. In 2010 4th International Universal Communication Symposium, pages 40–46. IEEE, 2010.

[31] Jeferson Santiago da Silva, François-Raymond Boyer, Laurent-Olivier Chiouette, and JM Pierre Langlois. External objects in p4: An rohc header compression scheme case study. In 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), pages 517–522. IEEE, 2018.

[32] Alexandru Seibulescu and Mario Baldi. Leveraging p4 flexibility to expose target-specific features. In Proceedings of the 3rd P4 Workshop in Europe, pages 36–42, 2020.

[33] Ouassim Karrakchou, Nancy Samaan, and Ahmed Karmouch. Endn: An enhanced ndn architecture with a p4-programmable data plane. In Proceedings of the 7th ACM Conference on Information-Centric Networking, pages 1–11, 2020.

[34] Hasanin Harkous, Michael Jarschel, Mu He, Rastin Pries, and Wolfgang Kellerer. P8: P4 with predictable packet processing performance. IEEE Transactions on Network and Service Management, 18(3):2846–2859, 2020.

[35] Frederik Hauser, Marco Hääberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. arXiv preprint arXiv:2101.10632, 2021.

[36] Debobroto Das Robin. An open source p416 compiler backend. https://github.com/drobinkent/P416CompilerBackend.

[37] Xiangyu Gao, Taeyyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In Proceedings of the 18th ACM Workshop on Hot Topics in Networks, pages 150–160, 2019.

[38] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In Architectures for Networking and Communications Systems, pages 13–24. IEEE, 2013.

[39] Péter Vörös, Dániel Horpácsí, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. T4p4s: A target-independent compiler for protocol-independent packet processors. In 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), pages 1–8. IEEE, 2018.

[40] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 435–450, 2020.

[41] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. Journal of Algorithms, 51(2):122–144, 2004.

[42] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. SIAM Journal on Computing, 39(4):1543–1561, 2010.
A QoS-Modifier P4 Program

The P416 source code for the QoS-Modifier program. We have implemented the program in P414 language and used it in sec. 6.2. The P414 version of the program is compatible with the TDG mapper described in [18].

```p4
// Definition of the stateful memory (register) array
register <bit<8>, bit<7>> (32w128) ipv4_port_qos;
register <bit<8>, bit<7>> (32w128) ipv6_port_qos;

// Table Definition starts from here
action set_ipv4_ipv6_qos() {
    ipv4_port_qos.write((bit<32>hdr.control_packet.index, hdr.control_packet.ipv4_diffserv));
    ipv6_port_qos.write((bit<32>hdr.control_packet.ipv6_trafficClass);
}
```
B Example V1Model Hardware Specification

Following JSON instance is written according to the Hardware Specification Language presented in sec 3.3. All the examples presented in sec 5 and the evaluation of our compiler backend presented in sec. 6.2 is based on the V1Model hardware represented by this JSON instance. Attributes in this JSON instance are annotated with the notations used to specify the hardware resources of V1Model switch in sec. 3.

```json
{
  "Name": "RMTV1Model32Stage",
  "TotalStages": 32,
  "HeaderVectorSpecs": [
    {
      "BitWidth": 8,
      "Count": 64,
      "ALU": "8BitAlu"
    },
    {
      "BitWidth": 16,
      "Count": 96,
      "ALU": "16BitAlu"
    },
    {
      "BitWidth": 32,
      "Count": 64,
      "ALU": "32BitAlu"
    }
  ],
  "ParserSpecs": {
    "ParsingRate": (PRate): 60,
    "HeaderIdentificationBufferSize": (PBH): 48,
    "MaxIdentifiableHeader": (H): 4,
    "MaxMoveAheadBit": (PMA): 128,
    "TCAMLength": (PBW): 256,
    "TCAMLookupFieldCount": (fTW): 4,
    "TCAMLookupFieldWidth": (fBW): 1,
    "MaxExtractableData": (PED): 48
  },
  "StageDescription": {
    "Index": "0-31",
    "PerMATICInstructionMemoryCapacity": (AC): 32,
    "ActionCrossbarBitWidth": (ACBW): 1280,
    "SRAMResources": {
      "MemoryPortWidth": (MBW): 80,
      "MemoryPortCount": (MP): 8,
      "MemoryBlockCount": (S): 106,
      "MemoryBlockBitWidth": (SBW): 80,
      "MemoryBlockRowCount": (SRL): 1024
    },
    "TCAMMatResources": {
      "MatchCrossbarBitWidth": (TCBW): 640,
      "BlockCount": (T): 16,
      "SupportedMatchTypes": {
        "exact": true,
        "lpm": true,
        "range": true,
        "ternary": true
      },
      "PerTCAMMatBlockSpec": {
        "TCAMBitWidth": (TBW): 40,
        "TCAMRowCount": (TL): 2048
      }
    },
    "SRAMMatResources": {
      "MatchCrossbarBitWidth": (SRBW): 640,
      "BlockCount": (S): 8,
      "SupportedMatchTypes": {
        "exact": true,
        "lpm": false,
        "range": false,
        "ternary": false
      },
      "PerSRAMMatBlockSpec": {
        "SRAMBitWidth": (SBW): 80,
        "HashingWay": (SH): 4
      }
    }
  }
}
```
"ExternResources": {
  "RegisterExtern": [
    {
      "name": "RegisterExtern_8"
    },
    {
      "name": "RegisterExtern_16"
    },
    {
      "name": "RegisterExtern_32"
    },
    {
      "name": "RegisterExtern_64"
    }
  ],
  "CounterExtern": [
    {
      "name": "CounterExtern_Packet"
    },
    {
      "name": "CounterExtern_Byte"
    },
    {
      "name": "CounterExtern_PacketByte"
    }
  ],
  "MeterExtern": [
    {
      "name": "MeterExtern_Byte"
    },
    {
      "name": "MeterExtern_Packet"
    },
    {
      "name": "RegisterExtern_32"
    }
  ]
}

"SingleStageCycleLength": 14,
"DependencyDelayInCycleLength": {
  "match_dependency": 12,
  "action_dependency": 3,
  "successor_dependency": 1,
  "reverse_match_dependency": 1,
  "default": 1
}