# Mapping Match+Action Tables to Switches

*Lavanya Jose*[⋆]*, Lisa Yan*[⋆]*, Pat Bosshart*[≀]*, Dan Daly*[†]*, George Varghese*[‡]*, Nick McKeown*[⋆]
[⋆]*Stanford University,* [≀]*Barefoot Networks,* [†]*Intel,* [‡]*Microsoft Research*

## 1 Background

In recent work, we proposed - with others - a new programming model for OpenFlow 2.0 [**?**]. We laid out three requirements: (1) Protocol independence, (2) Target independence - the programmer should write packet processing programs that can be compiled to a number of different target switches, (3) Reconfigurable in the field - it should be possible to change the way a switch processes packets. The forwarding model consists of a programmable parser followed by multiple match+action (arranged in series or parallel or a combination of both). The proposal in [**?**] is our starting point for the work presented here, and we assume the reader is familiar with the proposal.

To configure the switch, the programmer supplies a parse graph (to identify and type the header fields expected on the wire), a set of actions that are to be performed on packet headers, and a packet processing program, describing how packets are to be processed. We refer to these three pieces of information as the "configuration" and it is expressed using the P4 language defined in [**?**]. The P4 configuration determines which protocols are supported by the switch. Once the switch has been configured, the control plane can populate the configured match+action tables with forwarding rules as needed.

## 2 Problem Statement

In this work, we explore a mapping problem that must be solved in order to build a P4 compiler. P4 programs are written in terms of (logical) tables that hold forwarding rules, with associated actions. If the P4 program is to be target independent, then it must be written without reference to the number (and type) of forwarding tables in the physical switch. The logical tables defined by the programmer must be *mapped* to the physical tables on the switch.

The first task is to try and fit all the tables into the switch (which may have ordering dependencies constraining where they can be placed). More interestingly, we want to map the tables while meeting some objective functions, such as minimizing the number of stages used (to minimize latency) or maximizing the size of the unused tables (to make it easier to add new tables), and so on.

Our goal is to design a *mapper* that works with a variety of different switch architectures. In this paper, we will present preliminary results mapping tables to two very different architectures: The Intel FlexPipe [**?**] and the RMT pipeline [1].

The mapper obviously needs to know the number of physical tables in the switch and the type and size of different memory types available at each stage. Some switches (e.g., FlexPipe) may have parallel physical stages, thus the mapper needs to know which stages can execute simultaneously. Switches can also differ in how match words can be laid out across memory blocks (e.g., multiple words may be packed across one block in RMT but not in FlexPipe.) Additionally, depending on how the switches are implemented, there may be limits on how many match tables can be assigned to each stage or how many bits of the packet header can be matched in each stage. The mapper must take these important details into account as well.

The mapper much generate a mapping of logical tables to physical tables in different stages of the switch in a way that satisfies inter-table dependencies and additional switch specific constraints.

## 3 Approach

Our approach is to use integer linear programming (ILP) to express the mapping as an optimization problem, capturing the table sizes of the switch, the memory types, the dependencies between tables, and so on. We examine a number of different optimization objectives, such as minimizing the number of stages (to minimize latency).

### 3.1 Basic ILP

There are constraints on the total number of blocks used in each stage (capacity), on the order of tables (dependency) and on the number of match or action words that need to be assigned (assignment) to each stage. The compiler formulates these as an integer linear program. Here $Blocks_{mem,log,st}$ is the number of blocks of memory type *mem* assigned to logical table *log* in stage *st*. $Words_{mem,log,st}$ is the number of match words of table *log* that can be fitted in the *mem* blocks assigned to *log* in stage *st*. $Start_{log}$ and $End_{log}$ refer to starting and ending stages of logical table *log*.

$$\sum_{log} Blocks_{mem,log,st} \leq TotalBlocks_{mem,st} \ \forall mem, st \text{ (capacity constraint)}$$

$$\sum_{mem,st} Words_{mem,log,st} \geq TotalEntries_{log} \ \forall log \text{ (assignment constraint)}$$

$$\text{If } MatchDep_{log_b,log_a} > 0 \text{ then } End_{log_b} < Start_{log_a} \ \forall log_a, log_b \text{ (dependency constraint)}$$

### 3.2 Switch Specific details

The ILP formulation can be tailored to different switches - each with its own set of constraints. In this way, the same set of logical match+action tables can be mapped into different switch architectures relatively easily. We implement this using a switch specific preprocessor in our mapping program.

For example, note that the capacity and assignment constraint depend on how words can be laid out across memory blocks, which is a switch-specific feature. We let the switch specific preprocessor compute the sizes of different basic layouts (in memory blocks and number of match words) and input them to the compiler.

In the following $pf$ refers to the index of a particular layout of words of *log* across blocks of type *mem*.

$PfBlocks_{mem,log,pf}$ is the number of *mem* blocks required for a basic building block of the $pf$th layout. For e.g., one way to layout DstMac words (48 bits wide) across SRAM blocks that are 80 bits wide and 1000 entries deep is to pack five words evenly into three blocks. In this case a basic building block is made of $PfBlocks_{sram,DstMac,pf} = 3$ SRAM blocks and can fit $PfWords_{sram,DstMac,pf} = 5 \times 1000 = 5000$ DstMac addresses.

$BuildBlocks_{mem,log,st,pf}$ is the basic assignment variable which specifies how many building blocks of the $pf$th type (for $mem, log$) are assigned to *log* in stage *st*.

$$Blocks_{mem,log,st} = \sum_{pf} BuildBlocks_{mem,log,st,pf} \, PfBlocks_{mem,log,pf} \ \forall mem, log, st$$

$$Words_{mem,log,st} = \sum_{pf} BuildBlocks_{mem,log,st,pf} \, PfWords_{mem,log,pf} \ \forall mem, log, st$$

Constraints that are specific to only certain switches can be added to the basic program using assignment and starting and ending stage variables.

### 3.3 Different Objectives

ILP also lets us easily optimize for different user objectives. For example, to minimize the maximum number of pipeline stages used, we formulate the ILP by adding one constraint:

$$\text{if } \textstyle\sum_{mem,log} Blocks_{mem,log,st} > 0 \text{ then } maximumStage >= st. \text{ Then minimize } maximumStage.$$

Alternative objectives include minimizing the number of memory blocks used or minimizing the pipeline latency.

### 3.4 How good is the ILP Program?

Our ILP formulation scales *linearly* with the number of memory types, logical tables, dependencies and physical stages. For a realistic problem with, say, 10 logical tables, 20 pairwise dependencies among them and a switch with 32 stages of up to 106 SRAM blocks each, the ILP would need a few hundred variables and a few hundred constraints only. This is readily solved using a program such as CPLEX [**?**].

## References

[1] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 99–110.