A Generalized Algorithm for Flow Table Optimization

Charles Marsh
with David Walker (Advisor)
Department of Computer Science
Princeton University
{*crmarsh, dpw*}*@cs.princeton.edu*

Abstract

Network switches are both expensive and heterogeneous. The first property has pushed researchers to find ways to minimize or optimize the *flow tables* that store information in switches, producing solutions like those of Liu[10] and Meiners[18]. However, the second property (i.e., the diversity of switch hardware used in practice) has forced programmers to focus on the minute details of specific switches and, in some cases, rendered the algorithms developed by researchers far less useful and even inapplicable. In this paper, we present an attempt to create a series of highly general algorithms for flow table optimization which are parameterized on user-provided hardware specifications. These algorithms allow programmers to implement and optimize policies on network switches regardless of: the number of tables available in each switch; the number of fields in each table; the type of pattern-matching performed on each field; and the widths of the fields involved.

1 Introduction

Networking

In the most basic sense, a network is a set of switches, connected to one another at specific ports. Switches operate by taking in packets, observing these packets' *header fields* (e.g., source IP address, source MAC address), and matching such fields against patterns (or *predicates*) stored in the switches' flow tables. Some action is then performed depending on the predicate

matched. The canonical actions are: forward the packet to another switch in the network (*Fwd*) and drop the packet from the network (*Drop*).

A {*predicate, action*} pair is referred to as a *rule*; a collection of rules is referred to as a *policy* or *rule table*. Rules can operate over four types of predicates:

1. **Exact**: In this case, each predicate is of the form $\{0,1\}^n$ (e.g., 1011, 1010).

2. **Ternary**: In this case, each predicate is of the form $\{0,1,*\}^n$, where $*$ represents a "don't care" bit (e.g., 1*10 covers both 1010 and 1110).

3. **Prefix**: In this case, each predicate is of the form $\{0,1\}^{n-k}*^k$ (e.g., 101* covers both 1011 and 1010).

4. **Range**: In this case, each predicate is of the form $[i,j]$, which matches every binary predicate whose integer representation falls between i and j (e.g., [3,5] covers 3, 4, and 5, which correspond to 011, 100, and 101, respectively).

Table 1: A side-by-side comparison

Range	Exact	Prefix	Ternary
[0,2]	{00,01,10}	{0*,10}	{01,*0}

To make things clearer, consider the policy outlined in Table 2, which acts as a white-list, dropping all rules that do not match the source and destination IP addresses specified by the top two rules (rules are checked in priority order from the top down; a packet can match just one rule).

Table 2: A sample policy on two fields

Source IP	Destination IP	
11*	11*	*Fwd*
00*	001	*Fwd*
***	***	*Drop*

Software Defined Networking

Within each switch, the *control plane* handles the implementation of a given policy, while the *data plane* performs the actual packet processing. In the case of Table 2, then, for each packet that enters the switch, the data plane goes down the list of rules and executes the action listed

in a given row if and only if both the source and destination IP addresses satisfy the predicates in that row.

Software Defined Networking (SDN) is an abstraction that aims to strip away the control plane from individual network switches and instead allow programmers to implement policies at a high level by communicating with a single network *controller*. Rather than communicate with each individual switch, SDN lets programmers install packet-handling rules over a distributed set of switches while avoiding low-level details [7]. OpenFlow, in turn, is a protocol for specifying and implementing policies within the SDN paradigm [14].

Figure 1: Traditional networking

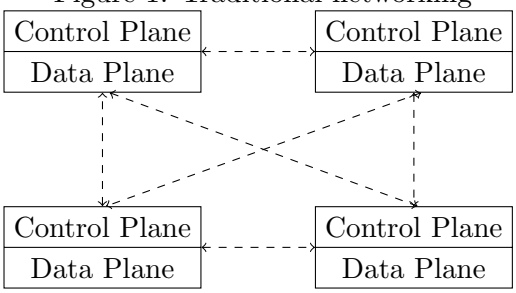
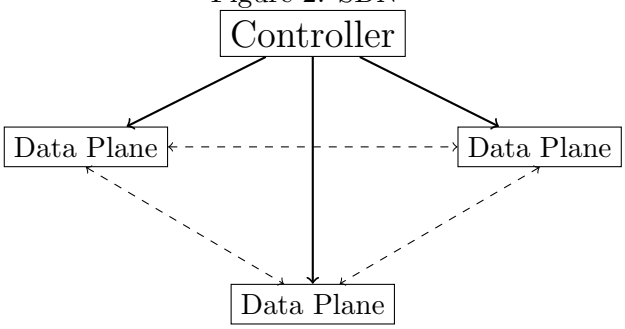


Figure 2: SDN



SDN is an extremely promising abstraction and its adoption has grown within industry [19]. However, many of today's controller platforms (OpenFlow included) still force programmers to focus on gritty, low-level implementation details, rendering their solutions both non-modular and tied to specific hardware. The canonical example of such non-modular behavior is OpenFlow's handling of policy composition (i.e., adding both policy A and policy B to a single switch). Instead of merely specifying said policies, the programmer must implement them in a precise order and assign seemingly arbitrary priorities; failing to do so leads to unexpected behavior. Thus, recent efforts, such as the development of the Frenetic language [4], have aimed to provide programmers with higher-level tools to work with SDN.

Network Hardware: Variety

The variety (inconsistency, even) of networking hardware used in industry further exacerbates the problem of low-level policy management. More specifically, different chipsets provided by different hardware manufacturers may demand completely different values for such parameters as: number of tables in a chip; number of fields in a table; and type of predicate matching

performed in each table (e.g., ternary, exact).

For example:

1. The Broadcom Trident switch, which is considered "representative of 10 Gigabet Ethernet switches", uses two tables, one which performs exact match lookup on two fields (VLAN ID and destination MAC), and another which performs ternary matching [21].

2. The Cisco Nexus 5000 switch, which contains multiple tables that perform either prefix-based or ternary matching on a single or multiple fields [2].

This forces the programmer to deal with different hardware architectures depending on the chipset in-use, creating an unnecessary burden.

Network Hardware: Performance

To make matters worse, switch hardware is costly, and the power consumption generated by such devices can be prohibitively high [3]. Thus, the memory capacities of each chip on a switch can be quite low in practice. This is especially true for chips that operate on ternary or prefix-based rules which, in most cases, use Ternary Content Addressable Memories (TCAMs) to perform O(1) matching on multiple fields [10]. With that in mind, it would be ideal to implement a given policy with as few rules as possible. That is, if we could implement the same network behavior with fewer rules, such a minimized set would always be preferred, as it would require both less space and less power. This is the *flow table minimization* problem.

But the flow table minimization problem is complicated, again, by the varied nature of switch hardware (as seen in the Broadcom vs. Cisco example), as programmers must worry about the number of fields and types of predicates used in each table, tying their solutions to the hardware involved.

Literature Review

Research on flow table minimization has focused on three approaches [24]:

1. **Range Encoding:** Range encoding maps a set of ranges to a "prefix-friendly" domain, such as in the solution of Meiners[16], before converting from range to prefix (many policies, such as firewalls, are specified by ranges initially). In the current paper, we avoid such

approaches, as they require pre-processing of packets to map their fields to new domains; this imposes an additional requirement on switch hardware [24].

2. **Hardware Improvement:** Self-explanatory. Although generally not considered in this paper, we structure our approaches so as to avoid imposing extra burdens or requirements on switch hardware; indeed, the purpose of our algorithm is to be as accommodating as possible to existing hardware.

3. **Classifier Compression:** This is the approach we adopt, which is to find more efficient, compact representations of policies. Recently, this approach has been developed by Liu, Torng, and Meiners[15]. There have also been efforts by Wei[24] and McGeer[13] to map TCAM policies to the field of Boolean logic and employ logic optimization algorithms. However, such techniques are avoided in this paper, as they too require packet pre-processing.

Goals

The question posed by this paper is as follows: can we develop a general, parameterized algorithm which allows programmers to specify characteristics of a switch, as well as a policy to implement, and receive, in return, a rule set which matches the desired characteristics? And further, given the memory constraints of network chipsets, can we optimize such policies based on the types of predicates involved?

The solution we present allows programmers to ignore hardware formatting and predicate types while achieving a high compression ratio. Indeed, in some cases, our solution seems to outperform existing single-table compression techniques by splitting a policy across multiple tables. Further, programmers can use our solution to run the same programs on a range of hardware, requiring simply a change in function arguments to specify the target configuration.

In summary, then, this paper addresses two major objectives:

1. Remove any burden from the programmer posed by the wide variety of switch hardware.

2. Compress the number of rules necessary to implement given policy across network switches, regardless of the number of tables involved or the formatting demanded by hardware.

Contributions

In this report, we present a technique for generating and optimizing policies given a set of user-specified parameters that describe a switch. Namely, given a policy and:

1. The number of tables in a given switch.
2. The number of fields in each table.
3. The widths of each field.
4. The type of predicates used in each table (a single type or a mixed set of formats).

our algorithm generates and optimizes a set of rule tables to fit the scenario.

Further, we offer a proof-of-concept implementation of the aforementioned technique and all the algorithms described through several modules written in the OCaml language, located in the rule-opt repository on GitHub.[1] Building on this work, we create a network simulator and integrate our work with the Desmoines compiler [6].

Finally, we conclude the paper with some results of running our algorithms on sample rulesets generated by the ClassBench tool [23].

2 Compression algorithms

In this section, we discuss techniques for one- and multi-dimensional compression of flow tables that use a single, uniform predicate format, concluding with a technique for mixed predicate optimization. Each of these techniques will be utilized in our generalized algorithm.

Compressing Exact-Match Flow Tables

The simplest compression case is that of exact-match flow tables, as there is no room for compression: every possible predicate in the domain must be spelled out exactly in the flow table.

[1] https://github.com/frenetic-lang/rule-opt

Compressing Prefix-Based Flow Tables

The case of optimally compressing prefix-based flow tables in a single dimension has been solved through the use of dynamic programming. The solution, described by Suri[22], relies on the concept of *consistency*, defined as: for every predicate $p' \in p$, matching p' on R returns rule a (in other words, every sub-predicate of p yields the same action a). For example, the policy in Table 3 is consistent on $0**$ with rule *Drop*. However, it is not consistent on $***$, as some predicates that match $***$ return *Drop* (e..g, 000), while others return *Fwd* (e.g., 100).

Table 3: An inconsistent policy

F_1	
1**	Fwd
00*	Drop
01*	Drop

In brief, the algorithm runs as follows:

1. Let p be a prefix-based predicate of the form $\{0,1\}^{k-n}*^k = p'*^k$, and R a set of rules on which you're trying to optimize.

2. If p is *consistent* on R with action a, return the rule $\{p, a\}$.

3. If p is not consistent on R, split p into $p0 = p'0*^{k-1}$ and $p1 = p'1*^{k-1}$. Recursively optimize R over $p0$ and $p1$ and combine the solutions. This process reduces down to the base case of an exact predicate $\{0,1\}^n$.

This algorithm is extended by Liu et al[10] to optimize prefix-based flow tables on multiple fields, a technique known as TCAM Razor. TCAM Razor first adjusts the single-dimensional solution of Suri[22] to operate with actions of weighted cost. In other words: the in the solution above is implicitly defined as the minimum number of rules produced; instead Liu et al offer an algorithm in which optimization is defined as the rule table with the minimum total cost of its weighted actions.

Next, Liu et al attempt to minimize a flow table field-by-field by converting the table into a *Firewall Decision Diagram* (FDD). An FDD is essentially a tree in which the edges match patterns (every pattern must match exactly one edge for a given level) and each level of nodes represents a different field [9]. Then, they optimize from the leaves upwards as follows:

1. For each field, consider the level of the FDD which corresponds to that field.

2. Create a "virtual" one-dimensional prefix-based optimization problem by pairing every predicate out of the FDD with the sub-FDD to which it points. Label the cost of the action to be the number of leaves in the sub-FDD (in the base case where the sub-FDD is an action, the cost is fixed at 1).

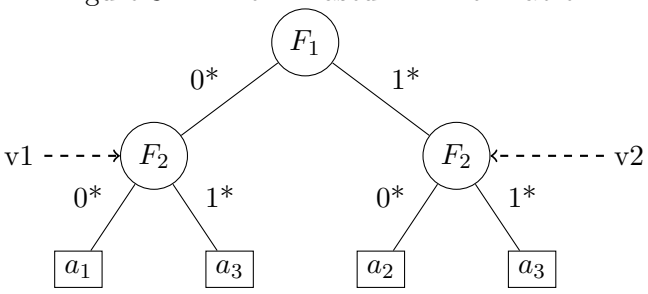
3. Optimize this weighted flow table and continue upward.

We include an example here, as the concept described above is essential to understanding the rest of the paper. Consider the policy in Table 4, for which an FDD is provided in Figure 3. TCAM Razor would operate by first performing one-dimensional compression on the leaf nodes of $v1$ and $v2$ separately; then, by performing the same compression on the root node with the two virtual actions "go to $v1$" and "go to $v2$".

Table 4: A sample policy

F_1	F_2	
0*	0*	a_1
1*	0*	a_2
**	**	a_3

Figure 3: A Prefix-Based FDD for Table 4



This idea of considering a set of non-terminal nodes to be "virtual" actions is essential to many of the compression algorithms described throughout the paper.

TCAM Razor then runs the output of its multi-dimensional optimization through a redundancy remover (more on this below). Note that for cases in which TCAM Razor returns a larger classifier than was input, the algorithm simply runs the redundancy removal technique and returns that output instead.

Compressing Ternary-Match Flow Tables

Compression of ternary-match rules is handled via a clever conversion from ternary- to prefix-based flow tables. In short, Liu et al devised an algorithm known as *bit weaving* in which they split flow tables into sub-tables and, within those sub-tables, perform a series of swaps to convert a set of ternary rules to prefix-based rules. For example:

$$swap_{2,4}(p_1 = 0 * 01, p_2 = 1 * 10) = (p'_1 = 010*, p'_2 = 101*)$$

These modified tables are then compressed with a one-dimensional compression algorithm, and the swaps are performed in reverse to output optimized flow tables [18].

A great advantage of this algorithm is that the multi-dimensional version merely requires us to concatenate our fields into a single field and, as the final step of the algorithm, split our predicates into multiple fields.

Compressing Range-Match Flow Tables

The solution to the range-match compression problem is described by Liu[12] and, similar to the prefix-based solution, uses dynamic programming techniques. To extend to multiple dimensions, the authors employ an identical technique to TCAM Razor in that they carry out level-by-level local optimization across the fields of the classifier.

Redundancy Removal

There are various redundancy removal techniques that can be implemented orthogonally to the above compression techniques (i.e., after compressing, one can run a flow table through a redundancy remover). There are two major types of redundancy [8]:

1. Upward: a rule r is upward redundant if there are no possible packets whose first matching rule is r. In Table 5, the final rule is upward redundancy, as any predicate of width three must match one of the first two rules.

2. Downward: a rule r is downward redundant if every packet that would match r would also match another rule r' with the same action as r. In Table 6, the middle rule is downward redundant, as any rule that would match the middle rule would also match the final rule, which has the same action.

Table 5: Upward redundancy

F_1	
1**	Fwd
0**	Drop
***	Fwd

Table 6: Downward redundancy

F_1	
1**	Fwd
00*	Drop
***	Drop

Compressing Mixed Flow Tables

The algorithm ("ACL-Compress") for compressing flow tables in which different fields have different predicate types is inspired by TCAM Razor and outlined by Liu et al[11]. To be completely clear, the difference between the problem statement here and any of the aforementioned compression cases is that in the previous cases, we had rules in which each predicate was of the same type. For example, in the rule $\{(F_1 = 00*, F_2 = 0**, F_3 = 100) \rightarrow Drop\}$, each field has a prefix predicate. However, we now examine cases in which the fields have varied types, such as the rule $\{(F_1 = [0, 17], F_2 = 0**, F_3 = 110) \rightarrow Drop\}$, which has $F_1 : range$, $F_2 : prefix$, $F_3 : exact$.

The key observation is that the level-by-level optimization approach used in TCAM Razor creates a scenario in which we can perform purely local optimizations in an FDD, ignorant of the optimization at any other level or even any other node. Thus, for any node n in the FDD with predicates of format f, we can run the one-dimensional rule optimization algorithm corresponding to f. Then, we can continue compressing as we move upward in the tree. Nowhere in the specification does it require that any two levels have the same predicate format, as all optimization is local.

Broadly speaking, the ACL-Compress algorithm thus takes the same structure as TCAM Razor; however, the single-node compression step is replaced with the following pseudocode:

```
let compress_at_node rules format =
  match format with
  | prefix_based -> one_d_prefix_compress rules
  | ternary -> bit_weave rules
  | exact -> enumerate_all rules
  | range -> one_d_range_compress rules
```

All of these algorithms are implemented in rule-opt (more on this later).

3 A Tagging Approach

We next examine how to split a single policy between multiple tables using *tags*. Specifically, we look at a scenario in which a user provides a policy P over multiple fields and desires, as output, multiple tables, each containing a subset of the fields in P. For example, we may take as input a table of the following form:

Src_IP	Dst_IP	Src_MAC	Dst_MAC	
...	...	...	...	Drop
...	...	...	...	Fwd

And, given some user specifications, output three tables (in order):

Src_IP	Dst_IP	
...	...	Tag 0
...	...	Tag 1

Tag	Src_MAC	
...	...	Tag 1
...	...	Tag 0
...	...	Tag 1

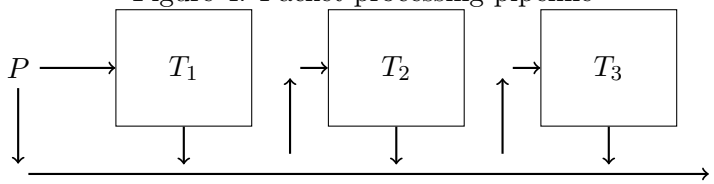
Tag	Dst_MAC	
...	...	Drop
...	...	Fwd

Notice that, within this multi-table framework, each table except the last has a 'Tag' column in-place of an 'Actions' column, which signifies adding some tagging bits to the packet for use in processing during the next table. More specifically, to process a packet p within this multi-table framework, we proceed as follows:

1. For each tagging table t_i:
   (a) Match p on the fields signified by t_i, including the tag appended by the previous table t_{i-1} (for the first table, the tag field is empty).
   (b) The action of t_i will be "*tag with* n" for some n. Replace p's current tag with n.
   (c) Pass p to the next table.
2. For t_{final}, process the packet p and return the resulting action.

What are the advantages of such an approach, and why is it worth examining? Firstly, such an approach is commonly used in industry, as suggested by Stephens et al[21], so its examination is evidently worthwhile. Secondly, creating such a *pipeline* for processing packets (as in Figure 4) poses great potential for parallelism: that is, if we split our single table into n tables, we

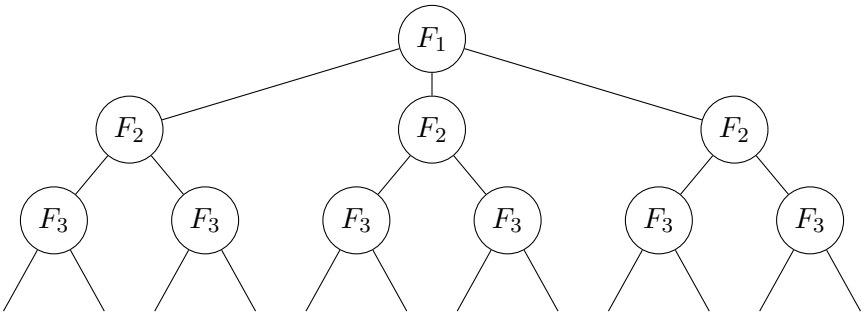
Figure 4: Packet processing pipeline



can process n packets in parallel, increasing our throughput. Additionally, breaking a multi-dimensional classifier into multiple, smaller classifiers can reduce information redundancy in TCAMs, which in turn allows us to use fewer rules to represent our policy [17].

Tagging a table

Our tagging algorithm ("Dynamic-Tag") relies on the use, once again, of FDDs. We first consider the problem of tagging a rule set based on a supplied FDD. To do so, we take as input an FDD and a list L of desired sizes for each output table. We assume that the FDD is constructed based on the order that the user desires his/her output fields to be present in its tables. That is, if the user wants two output tables, $T_1 : \{F_1, F_2\}$ and $T_2 : \{F_3\}$, we assume $L = [2; 1]$ and the FDD is of the following form:[2]



Given these inputs, our algorithm is outlined by the following pseudocode:

```
let tag_firewall fw specs =
  let n = size of next field defined by specs in
  let all_paths = bfs_traversal fw hd in
  let next_table = map (fun (path, fw') ->
    (prior_tag::path, tag_corresponding_to_fw')) all_paths in
  let rest = fold (fun path fw' acc ->
```

[2]Each node can have an arbitrary number of children, as each field can split on an arbitrary number of predicates.

```
    tag_firewall fw' tl tag_corresponding_to_fw') path in
  next::rest
let tag_table rules specs =
  let interval_firewall = construct_interval_firewall rules in
  let predicate_firewall = convert interval_firewall specs in
  tag_firewall predicate_firewall specs
```

For each desired output table with n fields, we run a BFS traversal from the current node to a depth of n, returning a list of edges and nodes at depth n. From this list, we construct a tag table by assigning each node its own tag and adding a rule for each edge that leads to that node.

Creating the FDD

We return to the problem of creating the FDD which will be input to the tagging algorithm. The backbone of the construction is described by Liu and Gouda[9]. However, the FDDs produced by this algorithm define their edges based on ranges. That is, the predicates on each edge of the FDD contain range intervals, rather than, say, prefixes.

To work around this issue, we need a set of efficient algorithms for range-to-prefix and range-to-ternary conversion. Such conversion is not trivial. Consider the case of expressing $[1, 14]$ in terms of prefixes; this requires six prefix-based predicates: $\{0001, 001*, 01**, 10**, 110*, 1110\}$.

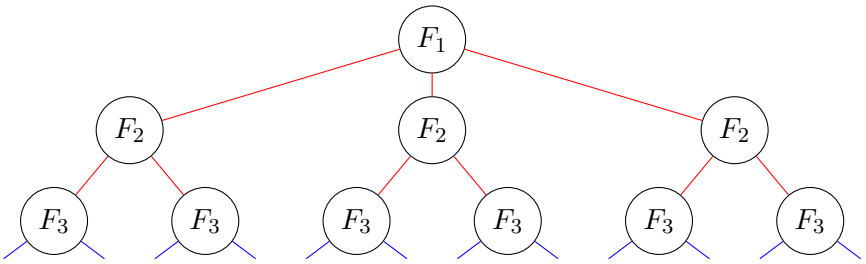
1. **Range-to-prefix conversion**: We use the DIRECT algorithm presented by Chang[1], which provides a worst-case expansion proportional to the binary logarithm of the size of the interval (i.e., for an interval $[1, 2^W - 2]$, DIRECT yields $2W - 2$ prefixes), and often performs much better.

2. **Range-to-ternary conversion**: We use the compute-dnf algorithm presented by Schieber et al[20], which always produces a set of ternary predicates of minimum cardinality.

3. **Range-to-exact conversion**: We merely enumerate every integer in the specified range.

With these algorithms, we can convert an entire FDD from range-based to any combination of predicate formats. That is, if the user wants as output two tables $T_1 : \{F_1, F_2\}$ and $T_2 : \{F_3\}$,

with T_1 using prefixes and T_2 using ternary-based predicates, we can convert the FDD to the appropriate formatting as follows (where red indicates prefix-based matching and blue indicates ternary matching):



The advantage of this approach is that, when we tag our tables using "Dynamic-Tag", we're ensured that the output tables will have the correct predicate formats, as the edges of the FDD from which the table rules are sourced will already be formatted correctly.

As a worked example, consider the FDD in Figure 5, which corresponds to the policy in Table 4 on page 8. If the user wanted to match on prefixes for F_1 and exact predicates for F_2, we would produce the modified FDD seen in Figure 6.

Figure 5: An Interval FDD for Table 4

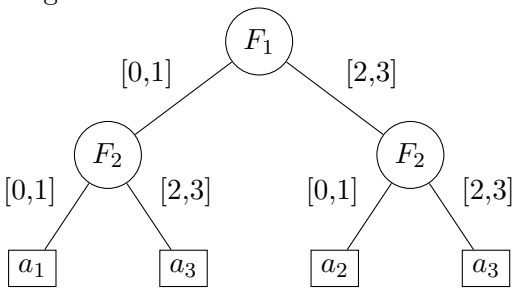
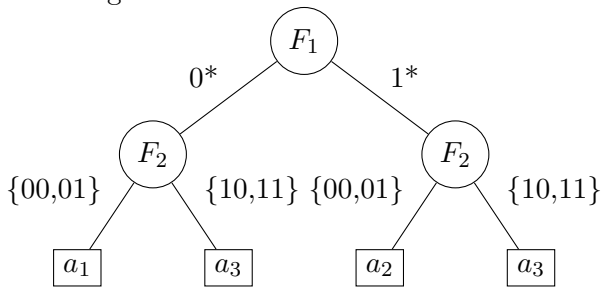


Figure 6: A mixed FDD for Table 4



Note that such an FDD can be constructed either:

1. From scratch to represent an entire policy.

2. Incrementally by adding one rule at a time to an existing FDD, as long as the rules are added *in priority order.*[3]

After construction, FDDs can be simplified (or *reduced*) using the FDD Reduction algorithm of Gouda and Liu[5]. This algorithm is implemented in rule-opt.

4 Tagging: A Worked Example

We now present a fully-worked example of the "Dynamic-Tag" algorithm, which produces a consistent set of flow tables based on user specifications. In the next section, we will focus on compressing tables produced by said algorithm.

Consider the policy P in Table 7. In this example, P is the input, along with the following parameters: the user would like T_1 to be a table on fields F_1 and F_2 that performs prefix-based matching; and the user would like T_2 to be a table on F_3 that performs ternary matching.

Table 7: Policy P

F_1	F_2	F_3	
101	000	01	*Fwd*
1**	1**	01	*Drop*
10*	111	00	*Fwd*
***	***	**	*Drop*

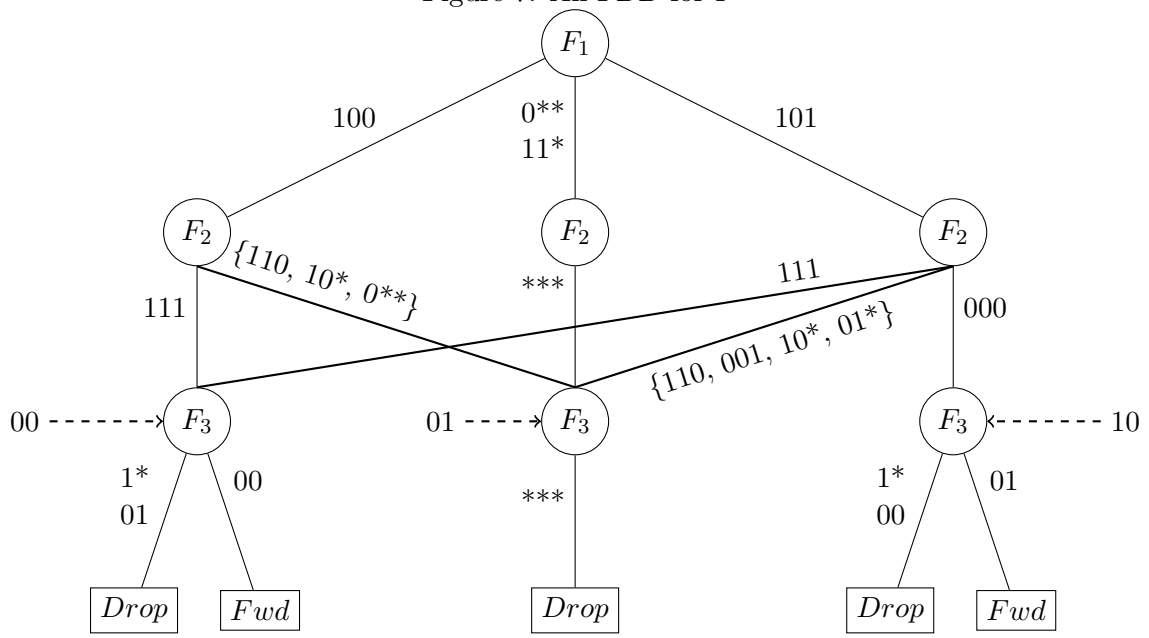
An FDD for P (following conversion to the appropriate formats and application of some FDD reduction techniques) is presented in Figure 7.

We now attempt to tag this FDD. Recall that the user specifications were for a table T_1 on fields F_1 and F_2, and a table T_2 on field F_3. Following "Dynamic-Tag", we run a BFS traversal to a depth of 2 (to generate T_1). This gives us three separate nodes, which we label with the tags 00, 01, and 10, as in Figure 7. For every path returned by BFS, we create a new rule for T_1 with the action "Tag with n", depending on which of 00, 01, or 10 were on the end of the path.

Next, from each of the nodes 00, 01, and 10, we run BFS to a depth of 1. For each path returned, we add a rule to T_2 with the action found at the leaf of the path; this rule will be prepended with the tag of the node at which the path began.

[3]We proceed on this assumption in our proof-of-concept implementation of the "Dynamic-Tag" algorithm; however, further work could be done to develop techniques for incremental FDD modification or deletion.

Figure 7: An FDD for P



For example, in the above FDD, following the right-most path down the tree, we'd get a rule $\{(F_1 = 101, F_2 = 000) \rightarrow$ "Tag with 10"$\}$ for T_1, and a rule $\{(Tag = 10, F_3 = 01) \rightarrow Fwd\}$ in T_2.

5 Compressing Tagged Tables

We now discuss compression techniques for our tagged tables. There are several approaches that we can take depending on the situation, the most practical and realistic of which are:

1. Taking a policy, decomposing it into several tables, and then compressing these tables individually.

2. Taking a policy, constructing the FDD for that policy, then compressing the FDD level-by-level and, finally, tagging it.

Specifically, with the first approach, we assume that the programmer would like to receive an uncompressed set of tables that mirror an input policy, with the ability to compress these tables later on; with the second approach, we assume the programmer would like to receive a set of compressed tables as the initial output. Both approaches are useful in different contexts. Thus, they are both supported in rule-opt.

In the first case (Approach 1), our compression algorithms will be of the form:

```
let tag_table rules specs =
  let interval_firewall = construct_interval_firewall rules in
  let predicate_firewall = convert interval_firewall specs in
  tag_firewall predicate_firewall specs

// called on every table produced by our tagging algorithm
let compress_table rules spec =
  match spec with
  | purely_prefix -> tcam_razor rules
  | purely_ternary -> multi_dimensional_bit_weave rules
  | purely_exact -> enumerate_all rules
  | purely_range -> multi_dimensional_range_compress rules
  | mixed -> acl_compress rules spec
```

In the latter case (Approach 2):

```
let tag_and_compress rules specs =
  let interval_firewall = construct_interval_firewall rules in
  let predicate_firewall = convert interval_firewall specs in
  let compressed_firewall = compress predicate_firewall specs in
  tag_firewall compressed_firewall specs
```

There's an advantage to Approach 2 which is only apparent when handling very large policies: with Approach 2, you avoid the excessive conversions present in Approach 1. That is, in Approach 1, if we want to tag then compress a rule table, we have to go from rule table to firewall to rule table to firewall to rule table. However, in Approach 2, we only have to make two such conversions (from rule table to firewall to rule table). The savings are very noticeable when working with large policies, especially as the firewalls produced by the FDD algorithm are generally much bigger than the rule tables on which they are based.

As a quick aside, we assume that "tag" fields are prefix-based. However, such fields are often *incomplete*, i.e., there are certain predicates for which there is no match. To be more specific, consider a policy which only ends up tagging its sub-nodes with tags '00' and '01'. Then, if you

try to match on a predicate with tag '11', there will be no matching rule. Compressing tables with this property is tricky, and to handle it with use a technique outlined by Liu et al[18]. In short, we consider a flow table R over k fields in which we have rules with actions $\{a_0, a_1, ..., a_n\}$. We add another rule to the end of R that spans the domain (i.e., $\{F_1 = *, F_2 = *, ..., F_k = *\}$) and has action a_{n+1}. With this rule, the flow table becomes complete. We compress this complete rule table; the resulting ruleset R' will have, as its final rule, $\{F_1 = *, F_2 = *, ..., F_k = *\}$ and action a_{n+1} due to the very large cost of a_{n+1}. We remove this final rule and take the remainder of R' as the compressed output.

6 The Rule-Opt Library

All of the work described above is implemented using the OCaml programming language, and can be found in the rule-opt library on GitHub. Here, we provide an outline of the library and the functionality it provides through the module-by-module rundown in Table 8.

Table 8: Rule-Opt Modules

Module	*Functionality*
Policy.ml	Basic definitions used throughout the library
Predicate.ml	Operations over prefix and ternary predicates
Interval.ml	Operations over intervals/range
Dynamic.ml	Handles single- and multi-dimensional prefix compression
Bitweave.ml	Handles single- and multi-dimensional ternary compression
Range.ml	Handles single- and multi-dimensional range compression
Redundancy.ml	Handles redundancy removal
Conversion.ml	A suite of methods for converting between predicate formats
Firewall.ml	Allows for construction and reduction of firewalls
Tag.ml	Implements the "Dynamic-Tag" algorithm
Compression.ml	Implements the generalized compression algorithms

Along with these modules are Testsuite.ml and Main.ml, both of which are compiled into executables. The former is used for unit-testing, while the latter is used for running compression simulations.[4]

We aimed to make the compression techniques in this library highly composable, as composing multiple techniques often leads to great savings. For example: our redundancy removal algorithm works on any type of input predicate (including ranges), and so can be added as a

[4]The numbers cited in the Results section are based on simulations run within this library.

final step in compression, orthogonal to other techniques. As an additional example of composability, in the *mixed_tree_compression* function of Compression.ml, we make the assumption that our flow tables are input with prefix-based predicates (there are similar functions to be used for cases in which you make different assumptions). Recall that in this function, we're running node-by-node compression based on the type of predicate used at the present node. Because we assume that input predicates are prefix-based, when compressing to ternary, we first run the optimal one-dimensional compression algorithm on the prefixes, and then compose this technique with bit weaving, earning some extra compression.

As a final note, we acknowledge that although this library includes a module for running range-based compression, such support is not available throughout. That is, the data type for predicates does not allow for mixing of range-based and other predicate formats. This is a function of the vast difference between a string of {0,1,*} and a range specified by two endpoints; thus, adding such support would have required much refactoring (although it is certainly possible).

7 Integration with the Desmoines Compiler

After developing "Dynamic-Tag" and the modified flow table optimization algorithms described above, we integrated the Rule-Opt library with the Desmoines Compiler, a framework for writing and running NetCore programs and simulations. Desmoines is written primarily in OCaml, so integrating our algorithms into the existing code base was a natural step.

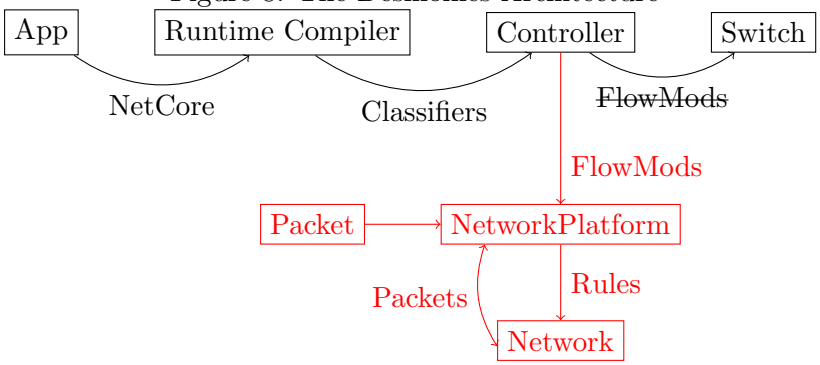
Desmoines: Form and Function

Desmoines provides a controller to handle software defined networks. That is, given some switches, Desmoines provides an interface for sending messages from a controller to those switches and vice versa. The most significant such message is a *flowMod*, which represents a request by the controller for a switch to modify its flow table.[5]

Desmoines uses the highly composable *NetCore* syntax to specify policies, which allows us to quickly develop complex and expressive network behaviors. The syntax itself is included in the Appendix.

[5] An example of a message from switch to controller is the case in which a switch does not know how to process an incoming packet and instead must send it to the controller for processing.

Figure 8: The Desmoines Architecture



Our use case

Desmoines allows programmers to communicate with existing switches. However, we wanted to provide some functionality by which we could simulate a switch's packet-processing capabilities. In effect, we wanted an *eval_packet* function which would allow us to send packets into a network and receive, as output, a list of packets that emerge from the switch. To do so, we had to create some additional modules, including:

1. *MultiTableNetwork.ml*: provides the abstraction for a set of switches (a 'network'), including data structures for holding flow tables for each switch and processing packets. This module could also call optimization functions from the rule-opt library.

2. *NetworkPlatform.ml*: encapsulates a network within a platform through which we can process messages between the controller and the rest of the network in an SDN style.

3. *NetworkPlatformTest.ml*: a series of tests run over the NetworkPlatform module.

The relationship between these modules and the rest of Desmoines is outlined in Figure 8. The typical Desmoines flow is in black, with our network simulator in red. Notice that our new modules allow us to plug in and evaluate packets. All packet evaluation and flow table optimization is performed by linking to the rule-opt library.

Implementing incremental policy creation

To fit within the Desmoines framework, we needed to allow for each switch to preform incremental updates to its policy. This is due to the fact that, within Desmoines, the controller

maintains ownership over a stream of policies, onto which flowMod messages are pushed and subsequently passed to their intended switches.

To that end, we had each switch maintain a copy of its FDD to-date. This allowed us to add rules to each switch's flow table with the following code:

```
let multi_switch = get_switch network id in
let firewall' = add_to_tree multi_switch.firewall rule widths in
let rule_tables' = tag_tree firewall' info num_fields in
let multi_switch' = { firewall'; rule_tables' } in
multi_switch'
```

Further, this avoided the unnecessary reconstruction of the FDD representing the switch's current flow table.

The successful running of our algorithm within the Desmoines framework, most importantly, provides a proof-of-concept. That is, integrating our code with Desmoines provides a practical look into how one could use such techniques as "Dynamic-Tag" to implement policies over switches in the real world. Additionally, the back-end we've developed for simulating packets across network switches may be useful as the Desmoines project continues to develop and new modules are written on top of the current platform. The full code can be found on GitHub in rule-opt/desmoines.

8 Results

We ran our algorithms on several sample policies generated by ClassBench, a tool for generating network policies that mirror those used in practice [23]. The use of such a tool is necessary, as no other large database of flow table policies are easily accessible due to security reasons.

Methodology

To start, we define the *size* of a rule table (the metric off of which we evaluate our algorithms), which is as follows: for a rule table R of n rules on k fields, each of which has width f_k, $size(R) = n * \Sigma_{i=0}^{k} f_k$. In other words, the size is the number of rules weighted by the width of each rule. The usefulness for this metric over, say, raw number of rules in a table, is that

often by splitting up a policy into multiple tables, we might increase the number of total rules spread over the output tables; however, each of these output tables will be on fewer fields, and often these fields will be of varying widths, all of which plays some role in the memory required to implement the table in practice. Therefore, we define our metric in such a way as to best mirror reality.

In order to run our simulations, we carried out the following process:

1. First, we generate a rule set using ClassBench. The sets generated by ClassBench consists of four fields: F_1 and F_2, which are prefix-based with widths of size 32; and F_3 and F_4, which are range-based with widths of size 16.

2. We next convert these ClassBench sets into purely prefix-based rule sets.

3. Finally, we run our compress_and_tag algorithm using example specifications. We assume that the fields retain their initial ordering (i.e., F_1, followed by F_2...), although this could be changed quite easily.

Data

Our results table (Table 9) is formatted as follows: In the first column is the name of the ClassBench seed file on which the rule set in question is based, suffixed with the number of rules output initially (e.g., 'fw1_60' indicates that the fw1 seed was used and 60 rules were output initially); next, we define P to be the ClassBench rule set following conversion to prefixes, and include the size of P (raw number of rules in parentheses). In addition, we include P', which is the result of running P under TCAM Razor, including the size of that table as well (again, with raw number of rules in parentheses) and the compression ratio achieved σ' (Size(P')/Size(P). After that, we layout a list of specifications for the output tables (e.g., [[T; T]; [P; P]] would indicate that we map to two tables, each of which has two fields, the first matching only on ternary and the second only on prefix-based rules); finally, we include the size of the output tables (summed) and the compression ratio σ_{Out}. We exclude the exact predicate format because of the ceiling it places on the amount of potential compression (and the range predicate for reasons described above).

Particularly noticeable in these results is that our algorithm is able to find some compression even when TCAM Razor utterly fails.

Table 9: Results of compress_and_tag

ClassBench	*Size(P)*	*Size(P′)*	σ'	*Output spec*	*Size(Out)*	σ_{Out}
fw1_60	8256 (86)	6336 (66)	0.769	[[P; P]; [P; P]]	3872 (90)	0.469
fw2_80	12480 (130)	12480 (130)	1	[[T]; [P]; [T]; P]	7504 (246)	0.602
fw3_80	17568 (183)	17568 (183)	1	[[P; P]; [T; T]]	10944 (282)	0.621
acl1_100	14400 (150)	12672 (132)	0.877	[[P]; [P; T]; [T]]	9168 (273)	0.637

9 Conclusion

To summarize, this paper presents a set of algorithms for performing generalized compression of policies in network switches. We contextualize our algorithms by acknowledging the diversity in network hardware, including the fact that different tables in different switches perform matching on different types of predicates. Along with the general concepts presented in this paper, we provide: 1) a library (rule-opt) that contains OCaml implementations of all the ideas and algorithms discussed, and 2) a network simulator, written to utilize the Desmoines compiler, in which the user can connect switches to a theoretical network, specify the format of their tables, and then optimize policies supplied in the NetCore language. In the end, we see these libraries and ideas as helpful in the broader context of creating high-level tools for SDN.

References

[1] Yeim-Kuan Chang. A 2-level TCAM architecture for ranges. *IEEE Transactions on Computers*, 55(12):1614–1629, 2006. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2006.189.

[2] *Understand ACL Resources on Cisco Nexus 5000 Switches.* Cisco, August 2011.

[3] Qunfeng Dong, Suman Banerjee, Jia Wang, Dheeraj Agrawal, and Ashutosh Shukla. Packet classifiers in ternary CAMs can be smaller. *SIGMETRICS Perform. Eval. Rev.*, 34(1):311–322, June 2006. ISSN 0163-5999. doi: 10.1145/1140103.1140313. URL `http://doi.acm.org/10.1145/1140103.1140313`.

[4] N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-

defined networks. *Communications Magazine, IEEE*, 51(2):128–134, 2013. ISSN 0163-6804. doi: 10.1109/MCOM.2013.6461197.

[5] Mohamed G. Gouda and Alex X. Liu. Structured firewall design. *Comput. Netw.*, 51 (4):1106–1120, March 2007. ISSN 1389-1286. doi: 10.1016/j.comnet.2006.06.015. URL `http://dx.doi.org/10.1016/j.comnet.2006.06.015`.

[6] A. Guha and M. Reitblatt. Desmoines, 2013. URL `https://github.com/frenetic-lang/desmoines`. Private repository.

[7] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing rule placement in software defined networks. Submitted for conference review, 2012.

[8] Alex X. Liu and Mohamed G. Gouda. Complete redundancy detection in firewalls. In *Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, DBSec'05, pages 193–206, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28138-X, 978-3-540-28138-2.

[9] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. *IEEE Trans. Parallel Distrib. Syst.*, 19(9):1237–1251, September 2008. ISSN 1045-9219.

[10] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: a systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 18(2):490–500, April 2010. ISSN 1063-6692.

[11] Alex X. Liu, Eric Torng, and Chad R. Meiners. Compressing network access control lists. *IEEE Trans. Parallel Distrib. Syst.*, 22(12):1969–1977, December 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2011.114. URL `http://dx.doi.org/10.1109/TPDS.2011.114`.

[12] A.X. Liu, E. Torng, and C.R. Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 176–180, 2008. doi: 10.1109/INFOCOM.2008.44.

[13] R. McGeer and P. Yalagandula. Minimizing rulesets for TCAM implementation. In *INFOCOM 2009, IEEE*, pages 1314–1322, 2009. doi: 10.1109/INFCOM.2009.5062046.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. ISSN 0146-4833.

[15] Chad R. Meiners, Alex X. Liu, and Eric Torng. *Hardware Based Packet Classification for High Speed Internet Routers*. Springer, 2010.

[16] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformations. In *Hardware based packet classification for high speed internet routers*, chapter 8, pages 101–119. Springer, 2010.

[17] Chad R. Meiners, Alex X. Liu, and Eric Torng. Sequential decomposition. In *Hardware based packet classification for high speed internet routers*, chapter 7, pages 75–101. Springer, 2010.

[18] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: a non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 20(2):488–500, April 2012. ISSN 1063-6692.

[19] Open Networking Summit. Software defined networking revolution, April 2013. URL `http://visual.ly/software-defined-networking-revolution`.

[20] Baruch Schieber, Daniel Geist, and Ayal Zaks. Computing the minimum dnf representation of boolean functions defined by intervals. *Discrete Appl. Math.*, 149(1-3):154–173, August 2005. ISSN 0166-218X. doi: 10.1016/j.dam.2004.08.009. URL `http://dx.doi.org/10.1016/j.dam.2004.08.009`.

[21] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. PAST: scalable ethernet for data centers. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1775-7.

[22] Suri Subhash, Tuomas Sandholm, and Priyank Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, April 2003.

[23] David E. Taylor and Jonathan S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007. ISSN 1063-6692. doi: 10.1109/TNET.2007.893156. URL http://dx.doi.org/10.1109/TNET.2007.893156.

[24] Rihua Wei, Yang Xu, and H Jonathan Chao. Block permutations in boolean space to minimize TCAM for packet classification.

Appendix

NetCore Policy Language

```
type predicate =
  | And of predicate * predicate
  | Or of predicate * predicate
  | Not of predicate
  | All
  | NoPackets
  | Switch of switchId
  | InPort of portId
  | DlSrc of Int64.t
  | DlDst of Int64.t
  | SrcIP of Int32.t
  | DstIP of Int32.t
  | TcpSrcPort of int (** 16-bits, implicitly IP *)
  | TcpDstPort of int (** 16-bits, implicitly IP *)


type action =
  | To of int
  | ToAll
  | GetPacket of get_packet_handler


type policy =
```

```
  | Pol of predicate * action list
  | Par of policy * policy (** parallel composition *)
  | Seq of policy * policy
  | Restrict of policy * predicate
  | Empty
```