Compiling Packet Programs to Reconfigurable Switches

Lavanya Jose and Lisa Yan, *Stanford University;*
George Varghese, *Microsoft Research;* Nick McKeown, *Stanford University*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose

This paper is included in the Proceedings of the
12th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Compiling Packet Programs to Reconfigurable Switches

Lavanya Jose*, Lisa Yan*, George Varghese‡, Nick McKeown*
*Stanford University, ‡Microsoft Research

Abstract

Programmable switching chips are becoming more commonplace, along with new packet processing languages to configure the forwarding behavior. Our paper explores the design of a compiler for such switching chips, in particular how to map logical lookup tables to physical tables, while meeting data and control dependencies in the program. We study the interplay between Integer Linear Programming (ILP) and greedy algorithms to generate solutions optimized for latency, pipeline occupancy, or power consumption. ILP is slower but more likely to fit hard cases; further, ILP can be used to suggest the best greedy approach. We compile benchmarks from real production networks to two different programmable switch architectures: RMT and Intel's FlexPipe. Greedy solutions can fail to fit and can require up to 38% more stages, 42% more cycles, or 45% more power for some benchmarks. Our analysis also identifies critical resources in chips. For a complicated use case, doubling the TCAM per stage reduces the minimum number of stages needed by 12.5%.

1 Introduction

The Internet pioneers called for "dumb, minimal and streamlined" packet forwarding [11]. However, over time, switches have grown complex with the addition of access control, tunneling, overlay formats, etc., specified in over 7,000 RFCs. Programmable switch hardware called NPUs [10, 17] were an initial attempt to address changes. Yet NPUs, while flexible, are too slow: the fastest fixed-function switch chips today operate at over 2.5Tb/s, an order of magnitude faster than the fastest NPU, and two orders faster than a CPU.

As a consequence, almost all switching today is done by chips like Broadcom's Trident [9]; arriving packets are processed by a fast sequence of pipeline stages, each dedicated to a fixed function. While these chips have adjustable parameters, they fundamentally cannot be reprogrammed to recognize or modify new header fields. Fixed-function processing chips have two major disadvantages: first, it can take 2–3 years before new protocols are supported in hardware. For example, the VxLAN field [21]—a simple encapsulation header for network virtualization—was not available as a chip feature until three years after its introduction. Second, if the switch pipeline stages are dedicated to specific functions but only a few are needed in a given network, many of the switch table and processing resources are wasted.

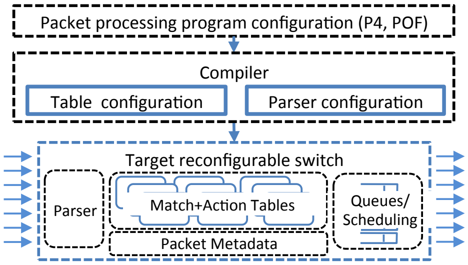


Figure 1: A top-down switch design.

A subtler consequence of fixed-function hardware is that networking equipment today is designed *bottom-up* rather than *top-down*. The designer of a new router must find a chip datasheet conforming to her requirements before squeezing her design bottom-up into a predetermined use of internal resources. By contrast, a top-down design approach would enable a network engineer to describe how packets are processed and adjust the sizes of various forwarding tables, oblivious to the the underlying hardware capabilities (Figure 1). Further, if the engineer changes the switch mid-deployment, she can simply install the existing program onto the new switch. The bottom-up design style is also at odds with other areas of high technology: for example, in graphics, the fastest DSPs and GPU chips [23, 28] provide primitive operations for a variety of applications. Fortunately, three trends suggest the imminent arrival of top-down networking design:

1. Software-Defined Networking (SDNs): SDNs [18, 24] are transforming network equipment from a vertically integrated model towards a programmable software platform where network owners and operators decide network behavior once deployed.

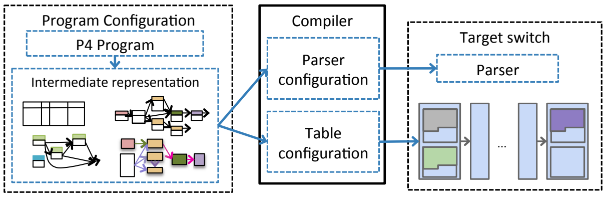


Figure 2: Compiler input and output.

2. Reconfigurable Chips: Emerging switch chip architectures are enabling programmers to reconfigure the packet processing pipeline at runtime. For example, the Intel FlexPipe [25], the RMT [8], and the Cavium XPA [4] follow a flexible *match+action* processing model that maintains performance comparable to fixed-function chips. Yet to accommodate flexibility, the switches have complex constraints on their programmability.

3. Packet Processing Languages: Recently, new languages have been proposed to express packet processing, like Huawei's Protocol Oblivious Forwarding [3, 27] and P4 [1, 2, 6]. Both POF and P4 describe how packets are to be processed abstractly—in terms of match+action processing—without referencing hardware details. P4 can be thought of as specifying control flow between a series of logical match+action tables.

With the advent of programmable switches and high-level switch languages, we are close to programming networking behavior top-down. However, a top-down approach is impossible without a *compiler* to map the high-level program down to the target switch hardware. This paper is about the design of such a compiler, which maps a given program configuration—using an intermediate representation called a Table Dependency Graph, or TDG (Section 2.1)—to a target switch. The compiler should create two items: a *parser configuration*, which specifies the order of headers in a packet, and a *table configuration*, which is a mapping that assigns the match+action tables to memory in a specific target switch pipeline (Figure 2). Previous research has shown how to generate parsing configurations [16]; the second aspect, table configuration, is the focus of this paper.

To understand the compilation problem, we first need to understand what a high-level packet processing language specifies and how an actual switch constrains feasible table configurations.

1.1 Packet Processing Languages

High-level packet processing languages such as P4 [6] must describe four things:

Abstract Switch Model: P4 uses the abstract switch model in Figure 1 with a programmable parser followed by a set of match+action tables (in parallel and/or in series) and a packet buffer.

Headers: P4 declares names for each header field, so the switch can turn incoming bit fields into typed data the programmer can reference. Headers are expressed using a parse graph, which can be compiled into a state machine using the methods of [16, 19].

Tables: P4 describes the *logical tables* of a program, which are match+action tables with a maximum size; examples are a 48-bit Ethernet address exact match table with at most 32,000 entries, or an 8K-entry table of 256-bit wide ACL matches.

Control Flow: P4 specifies the control flow that dictates how each packet header is to be processed, read, and modified. A compiler must map the program while preserving control flow; we give a more detailed example of this requirement in Section 2.1.

1.2 Characteristics of Switches

Once we have a high-level specification in a language, the compiler must work within the constraints of a target switch, which include the following:

Table sizes: Hardware switches contain memories that can be accessed in parallel and whose number and granularity are constrained.

Header field sizes: The width of the bus carrying the headers limits the size and number of headers that the switch can process.

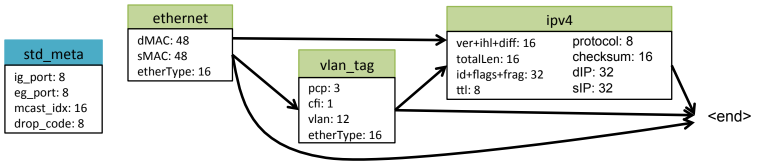
Matching headers: There are constraints on the width, format, and number of lookup keys to match against in each match+action stage.

Stage Diversity: A stage might have limited functionality; for example, one stage may be designed for matching IP prefixes, and another for ACL matching.

Concurrency: The biggest constraints often come from concurrency options. The three recent flexible switch ASICs (FlexPipe, RMT, XPA) are built from a sequential pipeline of match+action stages, with concurrency possible within each stage. A compiler must analyze the high-level program to find dependencies that limit concurrency; for example, a data dependency occurs when a piece of data in a header cannot be processed until a previous stage has finished modifying it. We follow the lead of Bosshart, et al. [8] and express dependencies using a TDG (Section 2.1).

1.3 Approach and Contributions

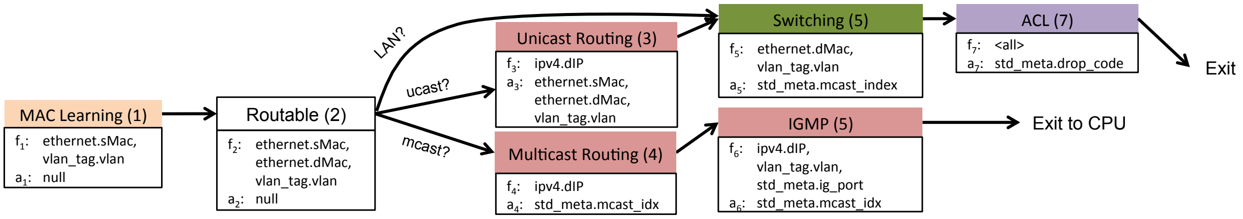
This paper is the first to define and systematically explore how to build a switch compiler by using abstractions to hide hardware details while capturing the essence required for mapping (Sections 2 and 3). Ideally we



(a) The parse graph specifies the order of the packet headers (green); the metadata (blue) is separate. All field names and lengths (in bits) are also specified.

Table name	l	Match type	e_l	n_l
MAC learning	1	exact	4000	(2)
Routable	2	exact	64	(3, 4, 5)
Unicast	3	prefix	2000	(5)
Multicast	4	prefix	500	(6)
Switching	5	exact	4000	(7)
IGMP	6	ternary	500	(to CPU)
ACL	7	ternary	1000	(exit)

(b) Each logical table l has a table name, maximum entry count e_l, and next table addresses n_l.


MAC Learning (1)
f_1: ethernet.sMac, vlan_tag.vlan
a_1: null
Routable (2)
f_2: ethernet.sMac, ethernet.dMac, vlan_tag.vlan
a_2: null
LAN?
ucast?
mcast?
Unicast Routing (3)
f_3: ipv4.dIP
a_3: ethernet.sMac, ethernet.dMac, vlan_tag.vlan
Multicast Routing (4)
f_4: ipv4.dIP
a_4: std_meta.mcast_idx
Switching (5)
f_5: ethernet.dMac, vlan_tag.vlan
a_5: std_meta.mcast_index
IGMP (5)
f_6: ipv4.dIP, vlan_tag.vlan, std_meta.ig_port
a_6: std_meta.mcast_idx
ACL (7)
f_7: <all>
a_7: std_meta.drop_code
Exit
Exit to CPU

(c) The control flow program. Each table l has match fields f_l and modified fields a_l.

Figure 3: A packet processing program named L2L3 describing a simple L2/L3 IPv4 switch.

would like a switch-dependent front-end preprocessor, and a switch-independent back-end; we show how to relegate some switch-specific features to a preprocessor. We identify key issues for any switch compiler: table sizes, program control flow, and switch memory restrictions. In a sense, we are adapting instruction reordering [20], a standard compilation mechanism, to efficiently configure a packet-processing pipeline. We reinterpret traditional control and data dependencies [20] in a match+action context using a Table Dependency Graph (TDG).

A second contribution is to compare greedy heuristic designs to Integer Linear Programming (ILP) ones; ILP is a more general approach that lets us optimize across a variety of objective functions (e.g., minimizing latency or power). We analyze four greedy heuristics and several ILP solutions on two switch designs, FlexPipe and RMT. For the smaller FlexPipe architecture, we show that ILP can often find a solution when greedy fails. For RMT, the best greedy solutions can require 38% more stages, 42% more cycles, or 45% more power than ILP. We argue that with more constrained architectures and more complex programs (Section 6), ILP approaches will be needed.

A third contribution is exploring the interplay between ILP and greedy, given ILP's optimal mappings despite its longer runtime. For each switch architecture, we design a tailored greedy algorithm to use when a quick fit suffices. Further, by analyzing the ILP for the "tightest" constraints, we find we can improve the greedy heuristics. Finally, a sensitivity analysis shows that the most important chip constraints that limit mapping for our benchmarks are the *degree of parallelism* and *per-stage memory*.

We proceed as follows. Section 2 defines the mapping problem and TDG, Section 3 abstracts FlexPipe and RMT architectures, Section 4 presents our ILP formulation, and Section 5 describes greedy heuristics. Section 6 presents experimental results, and Section 7 describes sensitivity analysis to determine critical constraints.

2 Problem Statement

Our objective is to solve the table configuration problem in Figure 2. We focus on mapping P4 programs to FlexPipe and RMT, while respecting hardware constraints and program control flow. Since the abstract switch model in Figure 1 does not model realistic constraints such as concurrency limits, finite table space, and finite processing stages, the compiler needs two more pieces of information. First, the compiler creates a table dependency graph (TDG) from the P4 program to deduce opportunities for concurrency, described below. Second, the compiler must be given the physical constraints of the target switch; we consider constraints for specific chips in Section 3.

2.1 Table Dependency Graph

We describe program control flow using an example P4 program called L2L3. Figure 3 describes the program by showing three of the four items described in Section 1.1: headers, tables, and control flow. The fourth item, the abstract switch model, is described in Section 3.

Our L2L3 program supports unicast and multicast routing, Layer 2 forwarding and learning, IGMP snooping, and a small Access Control List (ACL) check. Figure 3a is a parse graph declaring three different header fields (Ethernet, IPv4, and VLAN) and metadata used

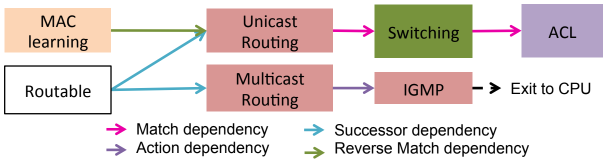


Figure 4: Table dependency graph for the L2L3 program.

Switch compiler	Traditional compiler
Match dependency	Read-After-Write
Action dependency	Write-After-Write
Successor dependency	Control dependence
Reverse-match dependency	Write-After-Read

Table 1: Mapping switch compiler dependencies to traditional compiler dependencies.

during processing. The features and control flow of the six logical tables in L2L3 are shown in Figure 3b and 3c.

Table l has attributes (f_l, e_l, a_l, n_l) that determine how a program should be allocated onto a target switch. A set of *match fields* f_l, from the packet header or metadata, are matched against e_l table entries. For example, the IPv4 Unicast routing table in L2L3 matches a 32-bit IPv4 destination address and holds up to 2,000 entries. In practice, table l may have much less than e_l entries, but the programmer provides e_l as an upper bound. Tables can have different *match types*: exact, prefix (longest prefix match), or ternary (wildcard). If the match type is ternary or prefix, the set f_l also specifies a bit mask. Based on the match result, the table performs actions on *modified fields* a_l and jumps to one of the tables specified in the set of *next table addresses*, n_l.

Figure 3c illustrates how header fields are processed by logical tables in an imperative control flow program. For example, the Unicast Routing table sets a new destination MAC address and VLAN tag before visiting the Switching table, which sets the egress port, and so on. The compiler must ensure that the matched and modified headers in each table correctly implement the control flow program.

We define a Table Dependency Graph (TDG) as a directed, acyclic graph (DAG) of the dependencies (edges) between the N logical tables (vertices) in the control flow. Dependencies arise between logical tables that lie on a common path through the control flow, where table outcomes can affect the same packet.

Figure 4 shows the TDG for our L2L3 program, which is generated directly from the P4 control flow and table description in Figure 3. From the next table addresses it is evident that some tables precede others in an execution pipeline; more precisely, Table A would precede Table B in an execution pipeline if there is a chain of tables $l_1, l_2, \ldots, l_k$ from A to B, where $l_1 \in n_A$, $l_2 \in n_{l_1}$, etc., and $B \in n_{l_k}$. If the result of Table A affects the outcome of Table B, we say that Table B has a *dependency* on Table A. In this case, there is an edge from Table A to B in the table dependency graph.

Different types of dependencies affect both the arrangement of tables in a pipeline and the pipeline latency.We present the three dependencies described in [8] and introduce a fourth below.

1. Match dependency: Table A modifies a field that a subsequent Table B matches.

2. Action dependency: Table A and B both change the same field, but the end-result should be that of the later Table B.

3. Successor dependency: Table A's match result determines whether Table B should be executed or not. More formally, there is a chain of tables $l_1, \ldots, l_k$ from A to B, where $l_1 \in n_A$, $l_2 \in n_{l_1}$, etc., and $B \in n_{l_k}$, such that every table $l_i \neq A$ in this chain is followed by B in each possible execution path. Additionally, there is a chain of next table adresses from A that does not go through B. For example, the Routable table's outcome determines whether Multicast Routing and IGMP will be executed. Thus, both have successor dependencies on Routable. On the other hand, IGMP does not have a successor dependency on Multicast Routing or vice-versa.

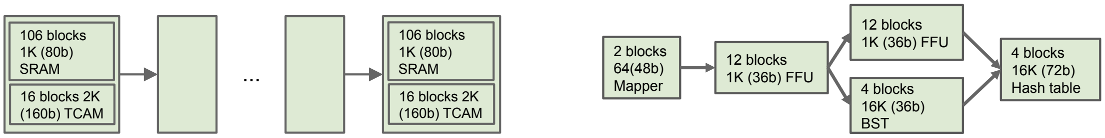
4. Reverse match dependency: Table A matches on a field that Table B modifies, and Table A must finish matching before Table B changes the field. This often occurs as in our example, where source MAC learning is an item that occurs early on, but the later Unicast table modifies the source MAC for packet exit.

Note that these dependencies roughly map to control and data dependencies in traditional compiler literature [5], where a match on a packet header field (or metadata) corresponds to a read and an action that updates a packet header (or metadata) corresponds to a write (Table 1).

While the TDG is strictly a multigraph, as there can be multiple dependencies between nodes, the mapping problem only depends on the strictest dependency that affects pipeline layout; the other dependencies can be removed to leave a graph. In summary, a TDG is a DAG of N logical tables (vertices) and dependencies (edges), where table $l \in \{1, \ldots, N\}$ has match fields, maximum match entries, modified fields, and next table addresses, denoted by f_l, e_l, a_l, and n_l, respectively.

3 Target Switches

The two backends we use—RMT [8] and Intel's FlexPipe [25]—represent real high-performance, programmable switch ASICs. Both conform to our abstract forwarding model (Figure 1) by implementing a pipeline of match+action stages and can run the L2L3 program in Figure 3. While both switches have different constraints, we can define hardware abstractions common to both chips: a pipeline DAG, memory types, and as-



(a) RMT switch as described in [8].

(b) Intel FlexPipe switch as described in [25].

RMT						
s	Mem. Name	Type	m	b_m	w_m	d_m
1-32	SRAM	exact	1	106	80b	1K
	TCAM	ternary	2	16	40b	2K

(c) Memory information for RMT.

FlexPipe						
s	Mem. Name	Type	m	$b_{s,m}$	w_m	d_m
1	Mapper	exact	1	1	48b	64
2-3	FFU	ternary	2	12	36b	1K
4	BST	prefix	3	4	36b	16K
5	Hash Table	exact	4	4	72b	16K

(d) Memory information for FlexPipe.

Figure 5: Switch configurations for RMT and FlexPipe. The tuple (s,m) refers to memory type m $(m \in \{1,...,K\})$ in the stage indexed by $s \in \{1,...,M\}$. Each (s,m) has attributes $(b_{s,m}, w_m, d_m)$, where $b_{s,m}$ is the number of blocks of the m-th memory type, and each of these blocks can match d_m words (the "depth" of each block) of maximum width w_m bits.

signment overhead. We describe these abstractions and switch-specific features, and highlight how our compiler represents each chip's constraints.

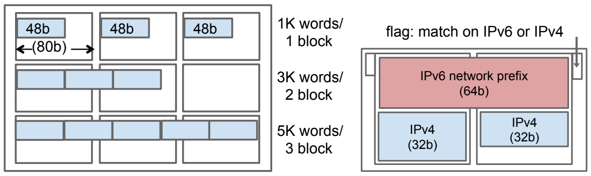
Pipeline Concurrency: We model the physical pipeline of each switch using a DAG of *stages* as shown in Figures 5a and 5b; a path from the i-th stage to the j-th stage implies that stage i starts execution before stage j. In the FlexPipe model (Figure 5b), the second Frame Forwarding Unit (FFU) stage and the Binary Search Tree (BST) stage can execute in parallel because there is no path between them.

Memory types: Switch designers decide in advance the allocation of different *memory blocks* based on programs they anticipate supporting. We abstract each memory block as having a *memory type* that supports various logical *match types* (Section 2.1). For example, in RMT, the TCAM allows ternary match type tables, while SRAM supports exact match only; in FlexPipe, FFU, hash tables, and BST memory types support ternary, exact, and prefix match, respectively.

Memory information for RMT and FlexPipe are in Tables 5c and 5d. We annotate the DAG to show the number, type and size of the memory blocks in each stage.

Assignment overhead: A table may execute actions or record statistics based on match results; these actions and statistics are also stored in the stage they are referenced. The number of blocks for action and statistics memory, collectively referred to as *assignment overhead*, is linearly dependent on the amount *match memory* a table has in a stage. In RMT, both TCAM and SRAM match memory store their overhead memory in SRAM; we ignore action and statistics memory in FlexPipe.

Combining entries: RMT allows a field to efficiently match against multiple words in the same memory block at a time, a feature we call *word-packing*. Different *packing formats* allow match entries to be efficiently stored in memory; for example, a packing format of 3 creates a *packing unit* that strings together two memory blocks and allows a 48b MAC address field to match against three MAC entries simultaneously in a 144b word (Figure 6a). FlexPipe only supports stringing together the minimum number of blocks required to match against one word, but does allow *table-sharing* in which multiple logical tables share the same block of SRAM or BST memory, provided the two tables are not on the same execution path. Table-sharing is shown in Figure 6b: since routing tables make decisions on either IPv4 or IPv6 prefixes, both sets of prefixes can share memory.



(a) Word-packing for SRAM blocks with $(w_m, d_m) = (80\text{b}, 1000)$.

(b) Table-sharing.

Figure 6: Block layout features in different switches.

Per-stage resources: RMT uses three crossbars per stage to connect subsets of the packet header vector to the match and action logic. Matches in SRAM and TCAM, and actions all require crossbars composed of eight 80b-wide subunits for a total of 640 bits. A stage can match on at most 8 tables and modify at most 8 fields. There appears to be no analogous constraints for FlexPipe.

Latency: Generally, processing will begin in each pipeline stage as soon as data is ready, allowing for overlapping execution. However, logical dependencies restrict the overlap (Figure 7). In RMT, a match dependency means no overlap is possible, and the delay between two stages will be the latency of match and action in a stage: 12 cycles. Action dependent stages can have

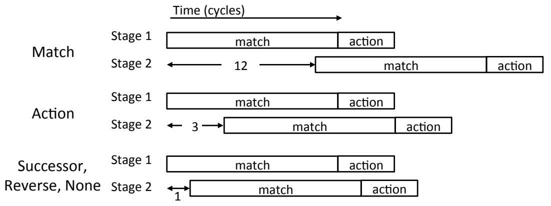


Figure 7: Dependency types and latency delays in RMT. In this figure, Table B in Stage 2 depends on Table A in Stage 1.

their match phases overlap, and so the minimum delay is 3 cycles between the stages. Successor and reverse-match dependencies can share stages, provided that tables can be run speculatively [8]. Note that even if there are no dependencies there is a one cycle delay between successive stages.

While RMT's architecture requires that match or action dependent tables be in strictly separate stages, FlexPipe's architecture resolves action dependencies at the end of each stage, and thus only match dependencies require separate stages. In summary, the compiler models specific switch designs abstractly using a DAG, multiple memory blocks per stage, constraints on packing, per-stage resources and latency characteristics. While we have described how to model RMT and FlexPipe (the only two currently published reconfigurable switches), new switches can be described using the same model if they use some form of physical match+action pipeline.

4 Integer Linear Programming

To build a compiler, we must map programs (parse graphs, table declarations and control flow) to target switches (modeled by a DAG of stages with per-stage resources) while maximizing concurrency and respecting all switch constraints. Because constraints are integer-valued (table sizes, crossbar widths, header vectors), it is natural to use Integer Linear Programming (ILP). If all constraints are linear constraints on integer variables and we specify an objective function (e.g., "use the least number of stages" or "minimize latency"), then fast ILP solvers (like CPLEX [12]) can find an optimal mapping.

We now explain how to encode switch and program constraints and specify objective functions. We divide the ILP-based compiler into a switch-specific preprocessor (for switch-specific resource calculation) and a switch-dependent compiler. We start with switch-independent common constraints.

4.1 Common Constraints

The following constraints are common to both switches:

Assignment Constraint: All logical tables must be assigned somewhere in the pipeline. For example, if a table l has $e_l = 5000$ entries, the total number of entries assigned to that logical table, or $W_{s,l,m}$ over all memory types m and stages s, should be at least 5000. Hence, we require:

$$\forall l: \sum_{s,m} W_{s,l,m} \geq e_l. \tag{1}$$

Capacity Constraint: For each memory type m, the aggregate memory assignment of table l to stage s, $U_{s,l,m}$, must not exceed the physical capacity of that stage, $b_{s,m}$:

$$\forall s,m: \sum_{l} U_{s,l,m} \leq b_{s,m}. \tag{2}$$

We define the assignment overhead as $\lambda_{m,l}$, which denotes the necessary number of action or statistics blocks required for assigning one match block of table l in memory type m. Thus the aggregate memory assignment is the sum of match memory blocks $\mu_{s,l,m}$ and assignment overhead blocks:

$$U_{s,l,m} = \mu_{s,l,m}(1 + \lambda_{l,m}).$$

Dependency Constraint: The solution must respect dependencies between logical tables. We use boolean variable $D_{A,B}$ to indicate whether table B depends on A, and the start and end stage numbers of any table l are denoted by S_l and E_l, respectively. If table B depends on A's results, then the first stage of B's entries, S_B, must occur after the match results of table A are known, which is at the earliest E_A (tables are allowed to span multiple pipeline stages):

$$\forall D_{A,B} > 0: \quad E_A \leq S_B. \tag{3}$$

If A must completely finish executing before B begins (e.g., match dependencies), then the inequality in Equation 3 becomes strict.

4.2 Objective Functions

A key advantage of ILP is that it can find an optimal solution for an objective function. In the remainder of the paper we focus our attention on three objective functions.

Pipeline stages: To minimize the number of pipeline stages a program uses, σ, we ask ILP to minimize:

$$\min \ \sigma, \tag{4}$$

where for all stages s:

$$\text{If} \sum_{l,m} U_{s,l,m} > 0: \qquad \sigma \geq s.$$

Latency: We can alternatively pick an objective function to minimize the total pipeline latency, which is more involved. Consider RMT, in which match and action dependencies both affect when a pipeline stage can start (whereas successor and reverse-match dependencies do not affect when a stage starts). If a table in stage s has a match or action dependency on a table in stage s', then s' cannot start until 12 or 3 clock cycles, respectively, after s. Building on how we expressed dependencies in Equation 3, we assign stage s a start time, t_s, where t_s is strictly increasing with s. Now consider two tables A and B, and assume B has a match dependency (i.e. 12 cycle wait) on table A. E_A is the last stage A resides in, and S_B is the first stage B resides in. We constrain S_B as follows:

$$t_{E_A} + 12 \le t_{S_B}.$$

We write the same constraints for all pairs of tables with action dependencies (3 cycle wait). Then we minimize the start time of the last stage, stage M:

$$\min \ t_M. \tag{5}$$

Power: Our third objective function minimizes power consumption by minimizing the number of active memory blocks, and where possible uses SRAM instead of TCAM. The objective function is therefore as follows:

$$\min \ \sum_m g_m\Big(\sum_{s,l} U_{s,l,m}\Big), \tag{6}$$

where $g_m(\cdot)$ returns the power consumed for memory type m.

4.3 Switch-Specific Constraints

Our ILP model requires switch-specific constraints, and we push as many details as possible to our preprocessor.

RMT: We start with RMT's ability to pack memory words together to create wider matches. Recall from Section 3 that a packing format p packs together p words in a single wide match; $B_{l,m,p}$ specifies the number of memory type m blocks required for packing format p of table l. While $B_{l,m,p}$ is precomputed by the preprocessor from the widths of the table entries and memory blocks, the ILP solver decides the number of packing units $P_{s,l,m,p}$ for each stage. We can thus find the number of match memory blocks $\mu_{s,l,m}$ and number assigned entries $W_{s,l,m}$ for each stage:

$$\mu_{s,l,m} = \sum_{p=1}^{p_{max}} P_{s,l,m,p} B_{l,m,p}.$$

$$W_{s,l,m} = \sum_{p=1}^{p_{max}} P_{s,l,m,p} (p \cdot d_m),$$

where $p \cdot d_m$ is the number of table l's entries that can fit in a single packing unit of format p in memory type m.

Per-Stage Resource Constraints: We must incorporate RMT-specific constraints such as the input action and match crossbars. The preprocessor can compute the number of input and action subunits needed for a logical table as a function of the width of the fields on which it matches or modifies, respectively.

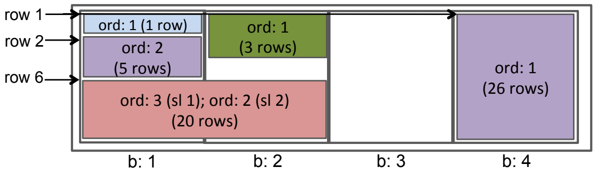


Figure 8: FlexPipe table sharing in detail. The pink table occupies the first two memory blocks, but different sets of tables share the first two memory blocks.

FlexPipe: FlexPipe can share memory blocks at a finer granularity than RMT, and so we need to preprocess the constraints differently for FlexPipe.

To support configurations as in Figure 8, we need to know which rows within a set of blocks are assigned to each logical table. This is because multiple tables can share a block, and different blocks associated with the same table can have very different arrangement of tables, such as blocks 1 and 2 assigned to the pink table.

Note that this issue does not arise in RMT; all memory blocks that contain a logical table will be uniform, and a solution can be rearranged to group together all memory blocks of a particular table assignment in a stage. We thus index the memory blocks $b \in 1, \ldots, b_{s,m}$, where $b_{s,m}$ is the maximum number of blocks of type m in stage s.

The solver decides how many logical table entries to assign to each block in each stage. For the remainder of this discussion, we differentiate between logical table entries and physical memory block entries by referring to the latter as rows, where row 1 and row e_m are the first and last rows, respectively, of a block of memory type m.

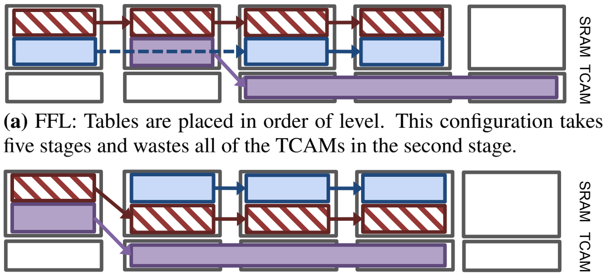
For table l assigned to start in the bth block of memory type m, we use the variable $\hat{r}_{l,m,b}$ to denote the starting row, and the variable $r_{l,m,b}$ to denote the number of consecutive rows that follow. [1]

To make sure rows do not overlap within a block, we constrain their placement by introducing the notion of *order*. Order is defined by the variable $\theta \in \{1, \ldots, \theta_{max}\}$, where θ_{max} is the maximum number of logical tables that can share a given memory block. In Figure 8, the light blue assignment has order $\theta = 1$, because it has the earliest row assignment. We define two additional variables, $\hat{\rho}_{m,b,\theta}$ and $\rho_{m,b,\theta}$, the start row and the number of rows of table with order θ, and we prevent overlaps by constraining the assignment as follows.

If θ-th order is assigned:

$$\hat{\rho}_{m,b,\theta-1} + \rho_{m,b,\theta} \le \hat{\rho}_{m,b,\theta}$$

[1] Note that if a second table, l', has entries in an adjacent block b', but the entries are wide and overflow into block b, $\hat{r}_{l',m,b} = 0$ because the starting row for l' was not assigned in this block; similarly, $r_{l,m,b}$ is irrelevant.



(a) FFL: Tables are placed in order of level. This configuration takes five stages and wastes all of the TCAMs in the second stage.

(b) FFLS: The first purple table with a large ternary table following it is placed first, even though the blue table has more match dependencies following it. This configuration uses only four stages.

Figure 9: Multiple-metric heuristics. A toy RMT example where a table with a single, dependent large ternary table must be placed before a table with a longer dependency chain.

To calculate the assignment constraint (Equation 1), the total number of words assigned to table l in stage s is:

$$W_{s,l,m} = \sum_{b=1}^{b_{s,m}} r_{l,m,b}.$$

where $r_{l,m,b}$ denotes the number of rows assigned for table l in all orders θ in block b of memory type m.

While the capacity constraint in Equation 2 is per stage, in FlexPipe we must also implement a capacity constraint per block. We restrict the number of rows we can assign to a block by checking the last row of the last order, θ_{max}:

$$\hat{\rho}_{m,b,\theta_{max}} + \rho_{m,b,\theta_{max}} \leq d_m.$$

Dependency Constraints: Fortunately, the dependency analysis is similar to RMT in Section 4.3, with the additional feature that only match dependencies require a strict inequality; action, successor, and reverse-match dependencies can be resolved in the same stage.

Objectives: Since FlexPipe has a short pipeline, we minimize the number of blocks used across all stages.

5 Greedy Heuristics

Since a full-blown optimal ILP algorithm takes a long time to run, we also explored four simpler greedy heuristics for our compiler: First Fit by Level (FFL), First Fit Decreasing (FFD), First Fit by Level and Size (FFLS), and Most Constrained First (MCF). All four greedy heuristics work as follows: First, sort the logical tables according to a metric. For each logical table in sorted order, pick the first set of memory blocks in the first pipeline stage the table can fit in without violating any capacity constraints, dependencies, or switch-specific resources. If it cannot fit, the heuristic finds the next available memory blocks, in the same stage or a subsequent stage. Like ILP, we leave switch-specific resource calculation like crossbar units and packing formats to a preprocessor. A heuristic terminates when all tables have been assigned or when it runs out of resources.

5.1 Ordering tables

The quality of the mapping depends heavily on the sort order. Three sorting metrics seem to matter most in our experiments, described in more detail below.

Dependency: Tables that come early in a *long dependency chain* should be placed first because we need at least as many stages left as there are match/action dependencies. We thus define the *level* of a table to be the number of match+action dependencies in the longest path of the TDG from the table to the end.

Word width: In RMT, tables with wide match or action words use up a large fraction of the *fixed resources* (action/input crossbars) and should be prioritized; they may not have room if smaller tables are assigned first. In FlexPipe, tables with larger match word width should be assigned first because there is less memory per stage.

Memory Types: While blocks with memory types like TCAM, BST, and FFU can also fit exact-match tables, exact memory like SRAM is generally more abundant than flexible memory due to switch costs. Thus in FlexPipe, heuristics should prioritize the assignment of more *restrictive tables*, or tables that can only go in ternary or prefix memories; otherwise, assigning exact match tables to flexible memories first can quickly lead to memory shortage. In RMT, restrictive tables go into TCAM, available in every stage. But large TCAM tables in a long dependency chain push back tables that follow them. So we should prioritize tables that imply high TCAM usage in their dependency chain.

Single-metric heuristics: Two of our greedy heuristics sort on a single metric: FFL is inspired by bin packing with dependencies [15] and sorts on table level, where tables at the head of long dependency chains are placed earlier. FFD is based on the First Fit Decreasing Heuristic for bin packing [15]. In our case, we prioritize tables that have wider action or match words and consequently use more action or input crossbar subunits. This heuristic should work well when fixed switch resources—and not program table sizes—are the limiting factor.

Multi-metric heuristics: Some programs fit well if we consider only one metric: if there are plenty of resources at each stage, we need only worry about long dependency chains. Our next two heuristics sort on multiple metrics. Sometimes being greedy on just one metric might not work, as shown in Figure 9: here, our first multi-metric heuristic FFLS incorporates dependencies and TCAM usage, where tables with larger TCAM tables in their dependency chains are assigned earlier.

Our other multi-metric heuristic, MCF, is motivated by FlexPipe's smaller pipeline with more varied memory types. We pick the "most constrained" table first: a table restricted to a particular memory type and with a

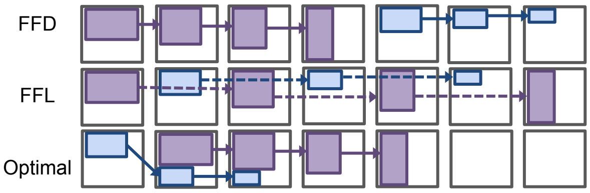
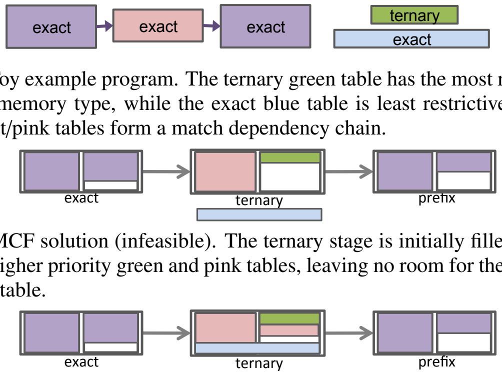


Figure 10: Greedy performing much worse than ILP (RMT). In this toy example, the blue and purple tables form separate match dependency chains. The initial blue table in the optimal mapping is narrower and has a lower level than the purple table, counterintuitive to both FFD and FFL metrics.



(a) Toy example program. The ternary green table has the most restrictive memory type, while the exact blue table is least restrictive. The violet/pink tables form a match dependency chain.

(b) MCF solution (infeasible). The ternary stage is initially filled with the higher priority green and pink tables, leaving no room for the wider blue table.

(c) Optimal solution. The pink table is split across ternary blocks, leaving enough room for the blue table.

Figure 11: Greedy performing much worse than ILP (FlexPipe).

Name	Switch	*N*	Dependencies		
			Match	Action	Other
L2L3 -Complex	RMT	24	23	2	10
L2L3 -Simple	RMT	16	4	0	15
	FlexPipe	13	12	0	4
L2L3 -Mtag	RMT	19	6	1	16
	FlexPipe	11	9	1	3
L3DC	RMT	13	7	3	1

Table 2: Logical program benchmarks for RMT and Flexpipe. *N* is the number of tables.

high level should have higher priority. Ties are broken by placing the table with wider match words first. FFL and FFD which ignore the memory type do not work well for FlexPipe, which does not have uniform memory layout per stage like RMT; for example, ternary match tables can only go in stages 2 or 3 in FlexPipe.

Variations: Each of the basic four heuristics has two variants: by default, an exact match table spills into TCAMs if it runs out of SRAMs in a stage. Our first variant prevents the spillage to preserve the TCAMs for ternary tables. Second, by default, when we reserve space for a TCAM table, we do not reserve space in SRAM to hold the associated action data, which means we may run out of SRAM and not be able to use the TCAM. Our second variant sets aside SRAM for action memory from yet-to-be allocated ternary tables; in our experiments we fix the amount to be 16 SRAMs.

There can be cases where the best combination of metrics is unclear, as in Figure 10 for RMT and Figure 11 for FlexPipe. Our experiments in Section 6 seek the right combination of metrics for an efficient greedy compiler.

6 Experiments

We tested our algorithms by compiling the four benchmark programs listed in Table 2 for the RMT and FlexPipe switches. The benchmarks are in Table 2: L2L3-Simple, a simple L2/L3 program with large tables; L2L3-Mtag, which is L2L3-Simple plus support for the mTag toy example described in [7]; L2L3-Complex, a complex L2/L3 program for an enterprise DC aggregation router with large host routing tables, and L3DC, which is a program for Layer 3 switching in a smaller enterprise DC switch. Differently sized, smaller variations of the L2L3-Simple and L2L3-Mtag programs are used for FlexPipe. L2L3-Complex and L3DC cannot run on Flexpipe because the longest dependency chain for each program needs 9 and 6 stages respectively, exceeding FlexPipe's 5-stage pipeline.

ILP: We used three ILP objective functions for RMT: number of stages (*ILP-Stages*), pipeline latency (*ILP-Latency*), and power consumption (*ILP-Power*). For FlexPipe, since we struggle to fit the program, we simply looked for a feasible solution that fit the switch. All of our ILP experiments were run using IBM's ILP solver, CPLEX. [2]

Greedy heuristics: For each RMT program, we ran all four greedy heuristics (FFD, FFL, FFLS, MCF). We also ran the variant that set aside 16 SRAM blocks for ternary action memory (labeled as FFD-16, etc.) and a combination of the two variants to also avoid spilling exact match tables into TCAM (labeled as FFD-exact16, etc.). For each FlexPipe program, we simply ran the greedy heuristic MCF. The other three heuristics do not combine enough metrics to fit either of our FlexPipe benchmarks.

All of our experiments were run on an Amazon AWS EC2 c3.4xlarge instance with 16 processor cores and 30 GB of memory. For FlexPipe, we generated 20 and 10 versions of the L2L3-Simple and L2L3-Mtag programs, respectively, with varying table sizes and checked how many greedy and ILP mappings fit the switch (Table 3). For RMT, we compiled every program 10 times for each of the greedy heuristics and the ILP objective functions

[2] CPLEX has a *gap tolerance* parameter, which sets the acceptable gap between the best integer objective and the current solution's objective. For ILP-Stage, we required zero-gap tolerance. For ILP-Latency and ILP-Power, we set the gap tolerance to be within 70% and 5%, respectively, of the best integer value; we found that lower gaps highly increased runtime with little improvement in objective value.

Solver	L2L3-Simple	L2L3-Mtag
	% solved	% solved
MCF	75	60
ILP	75	80

Table 3: Benchmark results for 5-stage FlexPipe.

Solver	L2L3-Complex			
	St.	Lat.	Pwr	RT.
FFD	22	135	4.98	0.25
FFD-16	21	135	5.51	0.27
FFD-exact16	21	135	4.62	0.27
FFL	19	131	5.61	0.25
FFL-16	19	131	6.09	0.27
FFL-exact16	17	132	4.61	0.24
FFLS	19	130	5.66	0.33
FFLS-16	19	130	6.42	0.35
FFLS-exact16	17	131	4.66	0.32
MCF	20	132	4.67	0.26
MCF-16	19	132	6.43	0.27
MCF-exact16	18	132	4.67	0.25
ILP-Latency	32	104	7.78	233.84
ILP-Stages	16	131	6.66	12.13
ILP-Power	32	131	4.44	147.10

Table 4: Benchmark results for RMT for L2L3-Complex. All greedy heuristics and variants are shown (St: number of stages occupied, Lat.: Latency [cycles], Pwr.: Power [Watts], RT.: Time to run solver [secs]).

and report the medians of the number of stages used, pipeline latency, and power consumed for each algorithm. We show in detail L2L3-Complex results in Table 4. To facilitate presentation, for all other programs we display results for ILP and the '-exact16' greedy variant only (Table 5), since this variant generally tended to have better stage and power usage than other greedy heuristics.

7 Analysis of Results

We analyze Tables 3, 4 and 5 for major findings. A salient observation is that the MCF heuristic for FlexPipe fits 16 out of the 20 versions of L2L3Simple. For some programs where the heuristic could not fit, it was difficult to manually analyze the incomplete solution for feasibility. However, ILP can both detect infeasible programs and find a fitting when feasible (assuming match tables are reasonably large,[3] e.g., occupy at least 5% of a hash table memory block.)

Another important observation for reconfigurable chips where one can optimize for different objectives, is that the best greedy heuristic can perform 25% worse on the objectives than ILP; for example, the optimal 104 cycle latency for ILP in the second column of Table 4 is far better than the best latency of 130 cycles by FFLS. A detailed comparison follows.

7.1 ILP vs Greedy

The following observations can be made after closely comparing ILP and greedy solutions in Figure 12.

1. Global versus local optimization: For the L2L3-Complex use case (Figure 12a), even the best greedy heuristic FFL-exact16 takes 17 stages, while ILP takes only 16 stages. Figures 12c and 12d show FFL-16 and ILP solutions, respectively. ILP breaks up tables over stages to pack them more efficiently, whereas greedy tries to assign as many words as possible in each stage per table, eventually wasting some SRAMs in some stages and using up more stages overall

In switch chips with shorter pipelines than RMT's, this could be the difference between fitting and not fitting. If all features in a program are necessary, then infeasibility is not an option. Unlike register allocation, there is no option to "spill to memory"; on the other hand, the longer runtime for ILP may be acceptable when adding a new router feature. Therefore it seems very likely that programmers will resort to optimal algorithms, such as ILP, when they really need to squeeze a program in.

2. Greedy poor for pipeline latency: Our greedy heuristics minimize the stages required to fit the program, and are good at minimizing power—the best greedy is only 4% worse than optimal (for L2L3-Complex, FFL-exact16 consumes 4.61W, versus ILP's 4.44W); technically, this is true only because the '-exact' variant avoids using power hungry TCAMs. But greedy heuristics are much worse for pipeline latency; minimizing latency with greedy algorithms will require improved heuristics.

7.2 Comparing greedy heuristics

1. Prioritize dependencies, not table sizes: In L2L3-Mtag, both FFL and FFLS assign the exact-match tables in the first stage, but differ in how they assign ternary tables. FFLS prioritizes the larger ACL table over the IPv6-Prefix and IPv6-Fwd tables, which are early tables in the long red dependency chain in Figure 13a. As a result, the IPv6-Prefix and IPv6-Fwd tables cannot start until stages 16 and 17, and FFLS ends up using two more stages than FFL. Though FFLS prioritizes large TCAM tables and avoids the problem discussed in Figure 9, it is not sophisticated enough to recognize other opportunities for sharing stages between dependency chains.

2. Sorting metrics matter: FFD results show that incorrect sorting order can be expensive (22 stages versus the optimal 16 for L2L3-Complex). We predict that FFD will only be useful for use cases with many wide logical tables or more limited per-stage switch resources, neither of which was a limiting factor in our experiments.

3. Set aside SRAM for TCAM actions: The '-16' vari-

[3] The minimum table size constraint helps us scale the ILP to handle FlexPipe, where a table can be assigned any number of rows in each memory block. Since the size of logical tables and memory blocks are at least on the order of hundreds, it seems reasonable to impose a minimum match table size of at least a hundred in these memory blocks.

Solver	L2L3-simple				L2L3-Mtag				L3DC			
	St.	Lat.	Pwr	RT.	St.	Lat.	Pwr	RT.	St.	Lat.	Pwr	RT.
FFD-exact16	21	64	7.54	0.18	22	75	7.65	0.21	7	88	2.34	0.08
FFL-exact16	19	55	7.55	0.19	19	66	7.66	0.21	7	88	2.34	0.08
FFLS-exact16	20	64	7.88	0.23	21	75	8.10	0.27	7	88	2.34	0.12
MCF-exact16	19	55	7.54	0.18	19	66	7.65	0.21	7	88	2.34	0.09
ILP-Latency	32	51	9.18	2.22	32	53	9.65	3.62	32	62	3.21	23.16
ILP-Stages	19	55	7.52	2.57	19	72	9.62	3.52	7	88	2.46	1.88
ILP-Power	32	62	7.55	2.27	32	71	7.63	2.53	9	86	2.34	1.87

Table 5: Benchmark results for 32-stage RMT for L2L3-simple, L2L3-Mtag, and L3DC. See Table 4 for units.

ation of our greedy heuristics estimates the number of SRAMs needed for ternary tables (for their action memory) in each stage and blocks them off when initially assigning SRAMs to exact match tables. Our experiments show that this local optimization usually avoids having to move a ternary table to a new stage because it doesn't have enough SRAMs for action memory.

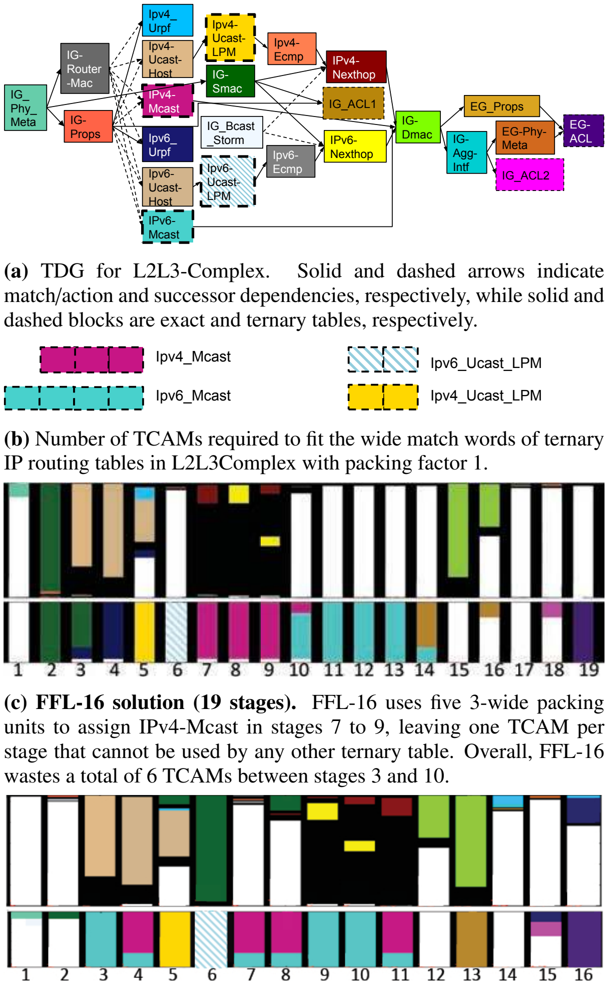
7.3 Sensitivity Experiments

In this section, we analyze ILP solutions by ignoring and relaxing various constraints in order to improve the running time of ILP and the optimality of greedy heuristics. We run these ILP experiments for our most complicated use case (L2L3_Complex) on the RMT chip, for two different objectives: minimum stages and minimum pipeline latency. For reference, the original ILP yields solution 16 stages in 12.13s and 104 cycles in 233.84s, respectively, for the two objectives.

To improve ILP runtime, we measure how long the ILP solver takes while ignoring or relaxing each constraint, which is a proxy for how hard it is to fit programs in the switch. This help us identify constraints that are currently "bottlenecks" in runtime for the ILP solver and also helps us understand how future switches can be designed to expedite ILP-based compilation.

We identify candidate metrics for greedy heuristics to optimize a given objective by ignoring constraints and identifying which have a significant impact on the quality of the solution. Our experiments also help identify the critical resources needed in the chip for typical programs, so chipmakers can design for better performance.

Sensitivity Results for ILP runtime: First, sizing particular resources can speed up IBM's ILP solver, CPLEX; for example, increasing the width of SRAM blocks by 37.5% (from 80b to 110b) reduces ILP runtime from 12.13s to 7.1s when minimizing stages. ILP run time is reduced considerably if action memory is not allocated, and this leads to a simple way to accelerate ILP: We first ran a greedy solution to estimate action memory needed per stage which is then set aside. We then ran the ILP without fitting the action memory, and finally added the action memory at the end. For minimizing pipeline latency, this reduced the ILP runtime from 233.84 sec-



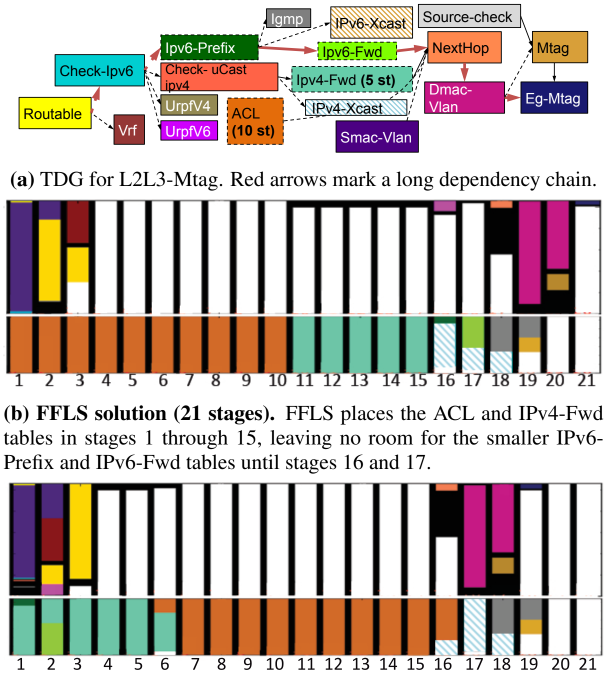
(a) TDG for L2L3-Complex. Solid and dashed arrows indicate match/action and successor dependencies, respectively, while solid and dashed blocks are exact and ternary tables, respectively.

(b) Number of TCAMs required to fit the wide match words of ternary IP routing tables in L2L3Complex with packing factor 1.

(c) FFL-16 solution (19 stages). FFL-16 uses five 3-wide packing units to assign IPv4-Mcast in stages 7 to 9, leaving one TCAM per stage that cannot be used by any other ternary table. Overall, FFL-16 wastes a total of 6 TCAMs between stages 3 and 10.

(d) ILP solution (16 stages). ILP utilizes all TCAMs in stages 4, 7, 8, and 11 by sharing the TCAMs between IPv4-Mcast (four 3-wide packing units) and IPv6-Mcast (one 4-wide packing unit).

Figure 12: FFL-16 and ILP solutions for L2L3-Complex. In assigning packing units to the ternary IP routing tables, FFL-16 locally maximizes the number of words per stage whereas ILP optimizes over a set of stages. Each stage has 106 SRAMs (top row) and 16 TCAMs (bottom row) and is colored according to the amount of match memory assigned to each logical table in the program TDG; all action memory is colored in black.



(a) TDG for L2L3-Mtag. Red arrows mark a long dependency chain.

(b) FFLS solution (21 stages). FFLS places the ACL and IPv4-Fwd tables in stages 1 through 15, leaving no room for the smaller IPv6-Prefix and IPv6-Fwd tables until stages 16 and 17.

(c) FFL solution (19 stages). FFL prioritizes the IPv6-Prefix and IPv6-Fwd tables and fits them in stages 1 and 2, allowing earlier assignment of Nexthop and other dependent tables. As a result, FFL needs two stages less than FFLS.

Figure 13: FFLS and FFL solutions for L2L3-Mtag. FFL prioritizes dependencies over table sizes and uses two fewer stages than FFLS.

onds to 66.29 seconds on average, without compromising our objective.

Sensitivity Results for optimality of greedy heuristics: We discovered that the dependency constraint for ILP has the largest impact on the minimum stages objective. If we remove dependencies from the TDGs, we can reduce the number of stages used from 16 to 13 and pipeline latency by 2 cycles. This explains why greedy heuristics focusing on the dependency metric (i.e., FFL and FFLS) do particularly well. Ignoring other constraints (like resource constraints) makes no difference to the number of stages used or latency. In addition, relaxing various resource constraints showed that some resources impact fitting more than others. For example, doubling the number of TCAM blocks per stage reduced the number of stages needed from 16 to 14. But doubling the number (or width) of crossbars made no difference. This explains why our FFD greedy heuristic (which focuses on non-limiting resources in the RMT switch) performs worse than other algorithms.

Lessons for chipmakers: Our results above indicate that chipmakers can improve turnaround for optimal ILP compilers by carefully selecting memory width. Moreover, if flexible memory is a rare resource, then increasing a non-limiting resource like crossbar complexity will not improve performance.

8 Related Work

Compiling packet programs to reconfigurable switches differs from compiling to FPGAs [26], other spatial architectures such as PLUG [14] or to NPUs [13]. We focus on *packing* match+action tables into memories in *pipelined stages* while satisfying dependencies. Nowatzki, et al. [22] develops an ILP scheduler for a spatial architecture that maps instructions entailed by program blocks to hardware, by allocating computational units for instructions and routing data between units. The corresponding problems for reconfigurable switches—assigning action units and routing data among the packet header, memories and action units—are less challenging once we have a table placement (and not in the scope of this paper.) NPUs such as the IXP network processor architecture [17] have multithreaded packet processing engines that can be pipelined. Approaches like that of Dai, et al. [13] map a sequential packet processing application into pipelined stages. However the processing engines have a large shared memory; thus NPU compilers do not need to address the problem of packing logical tables into physical memories.

9 Conclusion

We define the problem of mapping logical tables in packet processing programs. We evaluate greedy heuristics and ILP approaches for mapping logical tables on realistic benchmarks. While fitting tables is the main criterion, we also compute how well solvers minimize pipeline latency on the long RMT pipeline. We find that for RMT, there are realistic configurations where greedy approaches can fail to fit and need up to 38% more memory resources on the same benchmark. Three situations when ILP outperforms greedy are when there are *multiple conflicting metrics*, *multiple memory types* and *complicated objectives*. We believe future packet programs will get more complicated with more control flows, more different size tables, more dependencies and more complex objectives, arguing for an ILP-based approach. Further, sensitivity analysis of critical ILP constraints provides insight into designing fast tailored greedy approaches for particular targets and programs, marrying compilation speed to optimality.

Acknowledgements:

We thank Pat Bosshart and Dan Daly for many hours spent helping us understand RMT and Flexpipe. Many thanks also to Ravindra Sunkad and Changhoon Kim for providing us with detailed use cases. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-114747.

References

[1] P4 language spec version 1.0.0-rc2. `http://www.p4.org/spec/p4-latest.pdf`. Accessed: 2014-09-22.

[2] P4 website. `http://www.p4.org/`. Accessed: 2014-09-22.

[3] Protocol oblivious forwarding (pof) website. `http://www.poforwarding.org/`. Accessed: 2014-09-22.

[4] Xpliant packet architecture (xpa) press release. `http://www.cavium.com/newsevents-Cavium\-and-XPliant-Introduce-a-Fully-Programmable\-Switch-Silicon-Family.html`. Accessed: 2014-09-22.

[5] Appel, A. W. *Modern compiler implementation in C*. Cambridge university press, 1997.

[6] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[7] Bosshart, P., Daly, D., Izzard, M., McKeown, N., Rexford, J., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. Programming protocol-independent packet processors. *CoRR abs/1312.1719* (2013).

[8] 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.

[9] Broadcom Corporation. *Broadcom BCM56850 StrataXGS® Trident II Switching Technology*. Broadcom, 2013.

[10] Cisco Systems. *Deploying Control Plane Policing*. Cisco White Paper, 2005.

[11] Clark, D. The design philosophy of the darpa internet protocols. In *Proceedings of SIGCOMM 1988* (Cambridge, MA, Aug. 1988).

[12] CPLEX, I. I. 12.4, 2013.

[13] Dai, J., Huang, B., Li, L., and Harrison, L. Automatically partitioning packet processing applications for pipelined architectures. In *ACM SIGPLAN Notices* (2005), vol. 40, ACM, pp. 237–248.

[14] De Carli, L., Pan, Y., Kumar, A., Estan, C., and Sankaralingam, K. Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *ACM SIGCOMM Computer Communication Review* (2009), vol. 39, ACM, pp. 207–218.

[15] Garey, M. R., Graham, R. L., Johnson, D. S., and Yao, A. C.-C. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory, Series A 21*, 3 (1976), 257–298.

[16] Gibb, G., Varghese, G., Horowitz, M., and McKeown, N. Design principles for packet parsers. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on* (2013), IEEE, pp. 13–24.

[17] Intel Corporation. *Intel® Processors in Industrial Control and Automation Applications*. Intel White Paper, 2004.

[18] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 3–14.

[19] Kozanitis, C., Huber, J., Singh, S., and Varghese, G. Leaping multiple headers in a single bound: wire-speed parsing using the kangaroo system. In *INFOCOM, 2010 Proceedings IEEE* (2010), IEEE, pp. 1–9.

[20] Lam, M., Sethi, R., Ullman, J., and Aho, A. Compilers: Principles, techniques, and tools, 2006.

[21] Mahalingam, D., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and Wright, C. Vxlan: A framework for overlaying virtualized Layer 2 networks over Layer 3 networks. Internet-Draft draft-mahalingam-dutt-dcops-vxlan-00, IETF, 2011.

[22] Nowatzki, T., Sartin-Tarm, M., De Carli, L., Sankaralingam, K., Estan, C., and Robatmili, B. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices 48*, 6 (2013), 495–506.

[23] NVIDIA Corporation. *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}*. NVIDIA White Paper, 2009.

[24] Open Networking Foundation. *Software-Defined Networking: The new norm for networks*. Open Networking Foundation White Paper, 2012.

[25] Ozdag, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. *Intel Corporation* (2012), 8.

[26] Rinta-Aho, T., Nikander, P., Sahasrabuddhe, S. D., and Kempf, J. Click-to-netfpga toolchain.

[27] Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 127–132.

[28] Xilinx, Inc. *DSP: Designing for Optimal Results*. Xilinx, Inc., 2005.