Compiling Logical Packet Programs to Reconfigurable Switches: Table Mapping as a First Step

Abstract

Today's fixed-function switch chips result in new features taking years to develop, and in routers being designed bottom-up. This is changing with the emergence of software designed networks (SDNs), new chips such as Intels FlexPipe that allow flexible packet processing, and higher level languages for expressing packet processing such as P4. However, this vision requires compilers that translate logical packet processing specifications to low level switch configurations. Analogous to register allocation, our paper explores the problem of mapping logical lookup tables represented by a logical dependency graph to physical tables in a programmable switch. We investigate greedy algorithms based on bin-packing and more flexible approaches based on Integer Linear Programming (ILP). Key aspects of our solution are abstraction of hardware specific features, such as parallelism and memory technologies, and modularity. We test greedy and ILP approaches using benchmarks from real production networks, compiling to FlexPipe and RMT. Greedy approaches can fail and require 27% more resources on the same benchmark.

1 Introduction

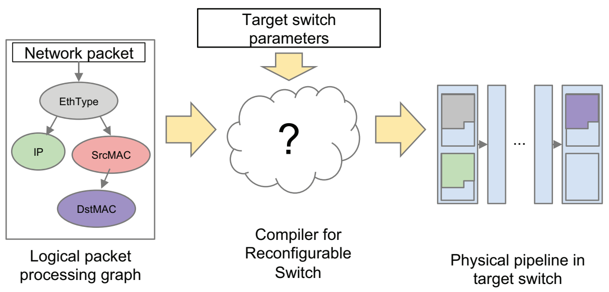

Target switch parameters
Network packet
EthType
IP
SrcMAC
DstMAC
Logical packet processing graph
?
Compiler for Reconfigurable Switch
...
Physical pipeline in target switch

Figure 1: The compiler lies between the hardware and logical specifications.

The Internet pioneers called for ”dumb, minimal and streamlined” packet forwarding [6]. However, over time, switches, routers and firewalls have become complicated with the addition of access control, tunneling, overlay formats, qualities of service, etc. specified in over 7,000 RFCs. To deal with constant change, developers tried creating programmable switch hardware, called NPUs [5, 8] consisting of many conventional CPUs on a single chip. NPUs have failed because they 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 fixed function chips such as Broadcoms Trident [3] where 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 add new header fields or set new header bits. Fixed 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 [10] - a simple encapsulation header for network virtualization - was not available as a chip feature until three years after its introduction.

A second subtler disadvantage of fixed function hardware is that networking equipment is designed ”bottom-up” rather than ”top-down”. The designer of a new router must find a chip datasheet conforming to her requirements, then squeeze her design into its pre-determined use of internal resources. The bottom-up design style is at odds with other areas of high technology. For example, in graphics, the fastest DSPs and GPU chips [11, 15] provide a set of primitive operations for a variety of applications and avoid hard-coding common tools like Fourier transforms. Fortunately, three new trends suggest the imminent arrival of top-down networking design:

1. SDNs: Software Defined Networks [9, 12] are changing network equipment from being vertically integrated towards a programmable software platform where network owners and operators decide network behavior once deployed.

2. Reconfigurable Chips: New switch chip architectures allow a programmer to reconfigure the pipeline to change packets processing at run-time, in the field. For example, the Intel Flexpipe [13] and the TI RMT [2] introduce a flexible "match-action" processing model without reducting performance over fixed-function chips. Both distinguish between *physical* match tables in the chips from *logical* match tables that the user specifies (see Figure 1). FlexPipe, the earlier chip, has certain stages dedicated to particular processing steps; RMT takes reconfigurability one step further with a protocol-independent pipeline built from primitive, protocol-agnostic building blocks. Both architectures have complex restrictions and many parameters to configure.

3. Packet Processing Languages: New languages have been proposed to express how packet processing such as Huaweis Protocol Oblivious Forwarding [14] and a recent strawman proposal called P4 [1]. Both POF and P4 are elegant languages that describe packet processing in abstract fashion without reference to the details of its implementation. P4 can be thought of as specifying a graph of logical tables with edges between nodes specifying dependencies.

Thus to get the full benefit from a high level language such as P4 and a flexible switch target like FlexPipe or RMT, we need to compile the user description to the target chip. A major subtask is to map logical tables to physical tables subject to target chip constraints and resources. A *correct* mapping will satisfy all switch constraints; however, an *optimal* mapping will minimize an objective function, such as table memory or latency. Our paper is the first study of algorithms to automate the mapping problem.

Consider the following analogy. When we write programs in Java, we do so without knowledge of the CPU the code compiles to. The programmer writes code using logical variables that a compiller maps to physical registers on the CPU. In traditional compilers, such register allocation is done using graph coloring [4], often approximated by a greedy algorithm. Similarly, switch compilers must map logical match tables to physical match tables but with the following differences. First, traditional compilers have a simple option on failure: spill a register to memory at some loss in performance. In a switch compiler, the cost of failure is much higher, because packets must be processed at line-rate; processing in software is often unacceptable. Second, register allocation has a simple and well-known objective function, whereas the objective functions are less obvious in a switch compiler. Third, traditional compilers must be very fast (msec) while switch compilers can be significantly slower (hours) as new switch functionality is not added lightly or often.

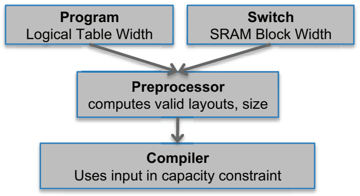


Figure 2: High-level system logic.

One approach is to design a different greedy algorithm to optimize for each new switch architecture. Instead, we seek a more general approach based on Integer Linear Programming (ILP). ILP allows us to express switch-specific constraints specific as well as a variety of objective functions with small changes to the code. While ILP can be thought of as a general hammer to attack resource allocation problems, ILP is akin to the Turing Tarpit : there are many ways to code resource constraints with little or no structure. ILP formulations can degenerate into spaghetti code inconsistent with the software engineering needs of a retargetable compiler.

A major contribution of our work is to show how ILP can be directly applied in modular fashion . The trick is to abstract away the differences between architectures to a general switch model; the abstractions we discuss in this paper are shown in Table 1. We argue that our models are flexible enough to extend to future chips. On the other hand, greedy approaches must be tailored per-architecture: greedy approaches for minimizing memory are naturally based on bin-packing while ones for minimizing latency are based on minimizing job completion times.

Further, our program is modularized into a packet processing model, a switch model, a switch-specific preprocessor, and a switch-independent compiler (see Figure 2). A number of switch-specific features can be relegated to the preprocessor. We do this for both ILP and greedy approaches to mapping and compare their efficacy on a set of benchmarks we gathered from various vendors.

Pipeline Feature	Abstraction
SRAM/TCAM/BST	Memory Types
Physical Parallelism	Topological State
Cross-memory layout constraints	Packing factors, preprocessor
Actions/Statistics	Duplicate resource constraint

Table 1: Abstractions for compiler model.

The paper makes the following contributions:

Problem: First paper to define and systematically explore the switch compiler mapping problem. (Section 2)

Abstractions: We define abstractions to hide hardware details and yet capture the essence required for mapping (Table 1, Section 3)

Experiments: We compare Greedy algorithms and ILP algorithms to perform the mapping optimized for latency, targeted at two different switch designs (FlexPipe and RMT). We show that ILP can fit designs where Greedy sometimes fails; using 27

The rest of the paper is outlined as follows. First, in Section 2 we define the mapping problem. Next, in Section 3, before exploring different types of solutions, we provide a set of abstractions upon which our solutions are based. Then in Section 4 and 5, we analyze heuristic and ILP approaches for two different switch architectures, the RMT chip and Intel FM 6000 FlexPipe chip. We demonstrate in Section 6 that ILP solutions are much more versatile and consistently produce optimal results, showing that there are packet processing tasks where ILP does 50% better than bin-packing. Finally, in Section 7 we discuss the implications of our approach in the context of the future of SDN.

2 Problem Statement

Our goal is to design a mapper that maps packet processing graphs (that represent desired packet computation) to a variety of different switch architectures, as shown in Figure 1. Such a mapper is a major component of a compiler that forms an intermediate layer between the user-specified control plane and the data plane implementation at the switch. We now precisely define models of packet processing graphs and switch architectures.

2.1 Packet Processing Graph

A packet processing graph input is a graph where the vertices are a set of N logical tables. Each logical table is indexed by $log \in \{1,...,N\}$ and is specified as $(m_{log}, e_{log}, a_{log}, nt_{log})$. Table log matches a set of fields m_{log}, either packet header fields or metadata and e_{log} denotes the number of entries in each table. Based on the result of the match, the table modifies a set of fields a_{log}, and jumps to processing the next table address that lies in the set nt_{log}. If there are n fields $m_1, \ldots m_n$ that are matched on or modified in any of the tables, the widths of all fields is also specified as input to determine the amount of memory the logical table will occupy. Matches can either be exact, prefix match, or ternary match. If the match is ternary, the set m_{log} also specifies a mask for each matched field.

For example, consider a simple L2/L3 IPv4 switch that supports unicast and multicast routing, Layer 2 forwarding and learning, IGMP snooping, and a small Access Control List (ACL) check. A sample set of logical tables

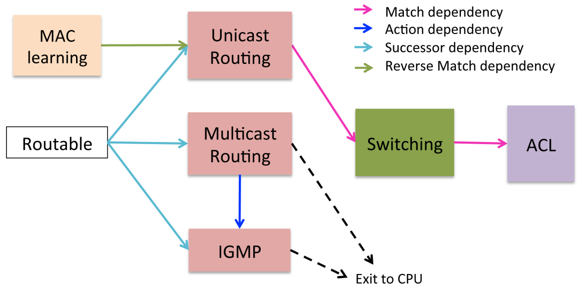


Figure 3: DAG of dependencies for a simple L2/L3 IPv4 switch.

that support these features are shown in Table 2. The Ipv4 Unicast routing table matches a 32bit IPv4 destination address and sets a new destination MAC address and VLAN tag, and has 16K entries. After setting these fields, the next table to be visited is the Switching table, which sets the egress port, and so on.

From the specified next table addresses it is evident that some tables must precede others in an execution pipeline. If the result of Table log_1 affects the outcome of Table log_2, we say that Table log_2 has a *dependency* on Table log_1. Thus, one can also generate a directed acyclic graph (DAG) of the dependencies between logical tables, as shown in Figure 3. Different types of dependencies affect both the arrangement of tables in a pipeline and the pipeline latency (Figure 4). We present the three dependencies described in [2] and introduce a fourth below.

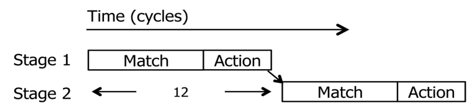
In a match dependency, Table log_1 modifies a field that Table log_2 matches on (Figure 4a). In our example, the Switching table has a match dependency on the Unicast table, which is an IP nexthop table that changes the destination MAC address. In an action dependency, Table log_1 and log_2 both change the same field, but the end-result should be that of the later Table log_2 (Figure 4b). For example, Multicast routing determines a set of multicast packets, but the set returned by IGMP is more specific, and so the final multicast packet set should be from IGMP, if available.

In a successor dependency, Table log_1 references Table log_2 as the next table address, and Table log_1 only executes as a result of this reference (Figure 4c). For example, only one of Unicast and Multicast is matched based on the outcome of the Routable table and hence both have a successor dependency with Routable. The new dependency that we introduce in this paper is a reverse-match dependency: Table log_1 matches on a field that Table log_2 modifies, and Table log_1 must occur before Table log_2 changes the field (Figure 4). 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

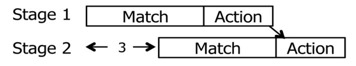
Table name	log	m_{log}	e_{log}	a_{log}	Next Tables (nt_{log})
MAC learning	1	(VLAN, src_MAC)	4000	null	(Routable (2))
Routable	2	(VLAN, dst_MAC, Ethtype)	64	null	(Unicast (2), Multicast (3))
Unicast	3	(dst_IP)	2000	(src_Mac, dst_MAC, VLAN)	(Switching (6))
Multicast	4	(dst_IP, VLAN)	500	(mcast_index)	(to CPU)
IGMP	5	(dst_IP, VLAN, ig_port)	500	(mcast_index)	(to CPU)
Switching	6	(dst_MAC, VLAN)	4000	(eg_port)	(ACL)
ACL	7	(all fields)	1000	(drop_code)	(exit)

Packet header			Metadata	
Ethtype (16b)	src_MAC (48b)	dst_MAC (48b)	ig_port (8b)	eg_port (8b)
VLAN (16b)	src_IP (32b)	dst_IP (32b)	mcast_index (16b)	drop_code (8b)

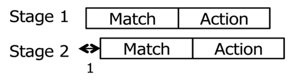
Table 2: A simple L2/L3 IPv4 switch.



(a) Match dependency.



(b) Action dependency.



(c) Successor, reverse match, or no dependency.

Figure 4: Latency execution, where Table log_2 is in Stage 2 and depends on Table log_1 in Stage 1. Cycle delays are specific to the RMT chip.

RMT						
Stage st	Mem. Type	Exact Match?	mem	B_{mem}	w_{mem}	d_{mem}
$1-32$	SRAM	✓	1	106	80b	1K
	TCAM		2	16	40b	2K

FlexPipe						
Stage st	Mem. Type	Exact Match?	mem	B_{mem}	w_{mem}	d_{mem}
1	Mapper	✓	1	1	48b	64
$2-3$	FFU		2	12	36b	1K
3	BST		3	4	36b	16K
4	Hash Table	✓	4	4	72b	16K

Table 3: Switch configurations for RMT and FlexPipe.

source MAC for packet exit.

While the graph of dependencies is strictly a multi-graph, as there can be multiple dependencies between nodes, the mapping problem only depends on the strongest dependency that affects pipeline layout, and the other dependencies can be removed to leave a graph.

In summary, a packet processing graph is:

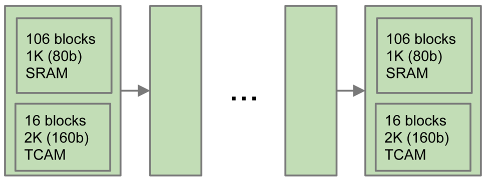
- a DAG of N logical tables (nodes) and dependencies (edges);
- for each logical table $log \in \{1,\ldots,N\}$, the fields to match on m_{log}, number of match entries e_{log}, fields to modify a_{log}, and next table addresses nt_{log};
- the widths of n fields modified or matched by any table.

2.2 Switch Model

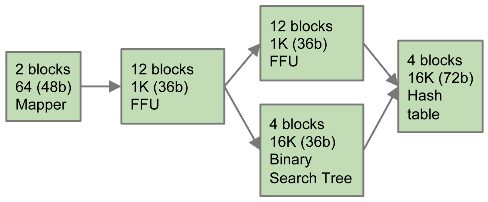
To define a generic mapping problem, we abstract common features of a switch architecture while allowing for switch-specific traits of the TI RMT [2] and the Intel FM6000 FlexPipe [13]. We abstract a switch as a physical pipeline modeled by a DAG of *stages*, each of which contains memory blocks of different *memory types* (Figure 5). If there is a path from the i-th stage to the j-th stage, then stage i must execute before stage j. This abstraction also enables us to model the parallel pipelines of FlexPipe chip (Figure 5a). As shown, the second Frame Forwarding Unit Block (FFU) and the Binary Search Tree Block (BST) can execute in parallel because there is no path between them.

More precisely, the physical pipeline supports k memory types spread across M stages. The tuple (st, mem) refers to memory type mem ($mem \in \{1,...,K\}$) in the stage indexed by $st \in \{1,...,M\}$. Each (st, mem) has an associated information tuple $(B_{st,mem}, w_{mem}, d_{mem})$, where $B_{st,mem}$ is the number of blocks of the mem-th memory type, and each of these blocks can match d_{mem} words (the "depth" of each block) of maximum width w_{mem} bits. The configurations for RMT and FlexPipe are shown in Table 3.

Each stage consists of two phases: In the match phase, packet fields and metadata are compared to stored entries



(a) RMT switch as described in [2].



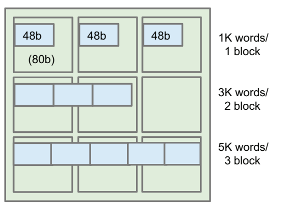
(b) Intel FlexPipe switch as described in [13]. The Hash Table represents SRAM used for exact matches, the BST represents an optimized longest prefix match memory, and the FFU represents a TCAM

Figure 5: Pipeline DAGs for RMT and FlexPipe.

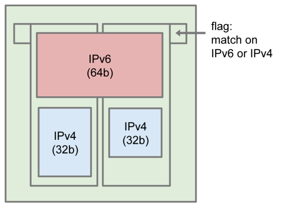
in memory blocks; all match results are compiled to resolve a list of actions to execute and next table addresses for future stages. The action phase executes the desired set of actions. Memory is used to store match options, action memory, and statistics; the memory to store instructions is often located outside of the main pipeline.

While the previously mentioned characteristics are common to all existing reconfigurable switches, each switch has particular features which include additional resource constraints per stage, flexibility in matching memory chunks to packet fields, and pipeline latencies.

The RMT chip uses three crossbars per stage to connect subsets of the packet header vector to the match and action logic. SRAM matches, TCAM matches, and action execution all require crossbars composed of 8 80b-wide subunits for a total of 640 bits. Each field occupies the number of 80b-subunits required to fit the field; for example, a 100b field requires 2 subunits. As a result, a stage can match on at most 8 tables and modify at most 8 fields. These per-stage resource constraints must also be specified as input. Based on the architecture we have seen, there are no analogous constraints for FlexPipe.



(a) Matching wide words: 48b MAC address can be packed together in different arrangements for SRAM with 1000 80-b entries.



(b) Sharing 32b memory blocks: packet routing is strictly IPv4 or Ipv6, so routing tables can share memory.

Figure 6: Block layout features in different switches.

Second, our model must also incorporate the means by which switch architectures support matching logical words wider than the internal widths of memory blocks. Two examples of this flexibility are depicted in Figure 6. The RMT chip allows a particular field to match against multiple words at a time. In the second row of Figure 6a, stringing two memory blocks together lets a 48b MAC address field match against three MAC entries simultaneously in a wider word of 144b. This so-called *word-packing* can reduce the wasted space that a logical table occupies in physical memory.

FlexPipe, on the other hand, can only support stringing together the minimum number of blocks required to match against one word. Instead, FlexPipe allows multiple logical tables to share the same block of SRAM or BST memory, provided that the two tables are not on the same execution path. This table-sharing functionality is shown in Figure 6b. Here, since routing tables make decisions strictly on either Ipv4 or Ipv6, the wider Ipv6 logical tables can share memory with Ipv4 tables.

Finally, we need to characterize the latency of each pipeline as part of the specification of a switch. In RMT, the match phase and action phase require 9 and 3 cycles, respectively, for a total of 12 cycles per stage. Generally, each stage will try to execute at the earliest timestep, allowing for overlap of stage execution. However, logical dependencies restrict this execution overlap, as in Figure 4: Table log_2 must operate on the results of Table log_2.

We first discuss in the context of RMT. For match dependencies, no overlap is possible, and the minimum delay between log_1's stages and log_2's stages is 12 cycles. For action dependencies, match phases can have overlap, leading to a minimum delay of 3 cycles. Successor and reverse-match dependencies can share stages, provided that tables can be run speculatively [2]. Note that even in the case of no dependencies between successive physical stages, there still exists a one cycle latency due to necessary packet forwarding delays between successive stages.

While RMT's architecture requires that match and action dependent tables be in strictly separate physical stages, FlexPipe's architecture resolves action dependencies at the end of each stage, and thus only match dependencies require a separation of stages.

In summary, every switch architecture is modeled by the compiler as:

- A directed, acyclic graph of M stages, where an edge between stage i and stage j imply that stage i must come before j in execution time
- For each memory type $mem \in \{1, \dots, K\}$ in stage $st \in \{1, \dots, M\}$, the number of memory blocks $B_{st,mem}$, where each memory block has width w_{mem} and maximum number of entries d_{mem}.
- Table layout constraints which determine how logical table entries can be packed across multiple blocks.
- Per-stage resource constraints such as input/action crossbar constraints and maximum tables to match
- Latency characteristics of pipeline execution

While we have built such models for RMT and FlexPipe (the only two publicly known reconfigurable switch today), it should be easy to build similar models for new switches provided that they use some form of physical pipeline.

3 Abstractions

The following abstractions hide the idiosyncracies of underlying switch architectures, while focusing on features that impact the mapping problem. This section elaborates on Table 1.

Memory types: Memory type distinctions arise due to manufacturing budgets; for example, TCAM, used for ternary matching like prefixes and wildcards, costs roughly 5× the transistor cost of SRAM (exact match) per bit. Switch designers decide in advance the allocation of different memory types to stages, based on the programs they anticipate supporting. For the purposes of our model, however, the salient features of a particular memory type are memory capacity (width and depth) and capability (exact match or ternary match), and the functionality details of each type are abstracted away. Our model can thus easily support new memory hardware additions.

Parallelism: Switches can have different topologies, with stages that execute sequentially or simultaneously. This is already abstracted in our switch model as a directed acyclic graph of stages. However, logical dependencies require assigning tables to stages that are in execution order. We introduce an abstraction to facilitate this assignment, where stages are numbered in order of their relative execution time.We implement this by topologically sorting the stages in the switch's pipeline.

Actions and Statistics: For each stage in the physical pipeline, match results are associated with different sets of actions. Each switch architecture specifies a bewildering variety of potential actions . Actions often require memory space to execute; if statistics are collected on match results, these also must be stored in memory. However, the number of blocks required for action memory or statistics storage linearly depends on the number of blocks assigned for matching to logical tables in each stage.

We therefore introduce another assignment called *assignment overhead*: the total number of blocks occupied by a logical table in a stage is a sum of the number of blocks required for matching words, executing actions, and storing statistics. The proportion of action and statistics memory per logical table assignment also becomes an input, abstracting away the implementation details.

Packing across memory blocks: We have seen that in both RMT and FlexPipe, memory blocks must be strung together to match against words that are wider than the width of a single block. We provide a flexible interface between memory block units and wide words by defining yet another abstraction: A *packing unit* is defined as a set of memory blocks that are strung together to match on a logical word of particular width.

For FlexPipe, it is suffices to have one packing unit format since FlexPipe does not support word packing. However, in RMT, the mapper can pick from several differently sized packing units for each table assigned in a stage. Hence, we define a packing unit for a logical table *log* and memory type *mem* as having two characteristic values: a format p, the number of words packed to form a wider word, and $BlocksOccupied_{mem,log,p}$, the number of type *mem* blocks it strings together to complete the match. For simplicity, we refer to a packing unit with format p as packing unit p.

Picking the best packing unit format can lead to more efficient use of memory blocks as shown in Figure 6a. In the figure, $p = 1$ leaves many unused bits in each memory block, whereas $p = 5$ ensures that the blocks can be completely filled width-wise. Yet always choosing the largest packing unit format can be wasteful: if there are only 1000 entries in this particular logical table, then $p = 1$ uses only one block, whereas $p = 5$ wastes a significant portion of 5 blocks. When assigning logical table *log* to memory type *mem* in stage *st*, a solution specifies the packing unit p and number of such packing units $PackingUnits_{st,log,mem,p}$, rather than the underlying memory blocks. $BlocksOccupied_{mem,log,p}$ is a function of p, *log*, and *mem* and can be precomputed to facilitate modularity, as we now discuss.

Modularity: We modularize the mapper into a switch-independent module which implements abstractions generated from a switch-specific preprocessor module. As we have seen, computing how much memory to assign to a logical table's entries depends on assignment overhead, word layout, and per-stage resource constraints, all switch-specific features. The switch-specific preprocessor determines how action and statistics memory sizes scale with match memory, so that the switch-independent module can refer to these variables as simple assignment overhead. Further, the preprocessor computes $BlocksOccupied_{p,log,mem}$ for different packing units p, tables log, and memory types mem, as well as resource limitations like input bar constraints in RMT.

4 Greedy Approaches

At its core, the mapping problem seeks to lay out logical tables into physical stages with resource constraints at each physical stage to optimize different objectives. In solving the mapping problem, there various tradeoffs between complexity, flexibility, scalability, and efficiency, both in terms of maximizing objective functions as well as minimizing running time. For example, an approach that uses many variables with respect to the number of logical tables and physical stages may be more complex but may also provide more flexibility. In this paper we seek approaches that not only work well on typical benchmarks but also scale well as the solution space increases.

The simplest approach is a greedy approach which can be summarized in the following two steps.

1. Sort tables according to some metric
2. Until tables are all assigned:
   (a) Pick first table available ready to be assigned
   (b) Calculate memory requirements
   (c) Put in first available place

Once a greedy heuristic picks a table *log* to assign, it considers whether assignment of the table in a set of blocks in stage *st* will violate any constraints of capacity, dependency, or switch-specific resources. When a table can fit in multiple memory types, we move to the next stage only if neither memory type works. If some constraint is violated, it moves onto the next set of available blocks, whether it be in the same stage or in the following stage. The greedy heuristics we consider continue in this manner until all tables have been assigned completely or it has run out of valid assignment areas across all stages.

For RMT, our first heuristic FFD is based on the First Fit Decreasing Heuristic for bin packing [7]. Tables are sorted in decreasing order of (most constrained) resource usage. Our second greedy heuristic is First Fit By Level (FFL) [7]. The tables are sorted in order of their level i.e., number of additional tables until the end of the control flow which corresponds to a level ordering in a topological sort of the graph. The intuition behind using bin packing heuristics is that when we assign a table to a stage (more specifically each packing unit of a table) it takes up a fixed fraction of resources available at that stage (e.g., number of input crossbar units to match words of a given width, number of action crossbar bits for action data, number of SRAM blocks.) Two very wide tables may not fit in the same stage (irrespective of the number of entries.) However there are quite a few variations from the typical bin packing problems- a table can use a variable number of RAM units in a stage, and if it is large enough its RAM units may spill over into other stages. Also, a table can use one or both of the memory types available in each stage. The resources constraints may be per memory (e.g., one input crossbar each for TCAM and SRAM) or for both memories (e.g., one action data crossbar for both TCAM and SRAM.) Despite the differences, our experiments show that bin-packing heuristics work reasonably well for minimizing the number of stages.

For FlexPipe, our heuristic sorts tables in decreasing order of how constrained they are. Then in each round, we pick and assign the first table that is ready to be assigned (i.e., does not depend on any table that hasn't been assigned yet). We place the table entries in contiguous slices from the earliest slice in the earliest stage possible. If at any point using a slice would violate some constraint (e.g., maximum tables per slice), we just switch to the next slice. Our exact metric for how constrained a logical table looks at how many of the five FlexPipe stages the table could possibly go in based on its memory type (e.g., ternary match can only go in Stage 2 or Stage 3, which have ternary match FFUs) and the length of the longest chain of dependencies following it (e.g., if there is a chain of three tables A, B, C like A $\rightarrow$ B $\rightarrow$ C where $\rightarrow$ is a match dependency, then A can't go in Stage 4 or 5, and must leave one stage each for B and C). For tables which have the same number of stages they could go in, we place the wider table first.

While heuristic approaches work effectively for a subset of logical table layouts, there are several cases where heuristics cannot even come close to a feasible, much less optimal, answer. In our ILP approach, we seek a reliable solver that consistently provides a correct and optimal mapping at the cost of possibly longer running times.

5 ILP Approach

In Integer Linear Programming (ILP), we write a set of linear constraints on integer variables and specify an objective function. In RMT only one logical table can occupy a memory block, so it is natural to use a variable that describes whether a logical table uses a block. However, in FlexPipe, logical tables can share a physical block so finer-grained variables are needed to record the number of rows assigned within a block to each logical table, and which row the assignment starts in.

Despite their differences, the two ILP approaches share the fundamental problem of assignment of part of a logical table to a particular physical stage. We thus first describe the constraints and objective functions common to these architectures and follow with switch-specific constraints.

5.1 Common Constraints

Assignment Constraint: All tables must be assigned somewhere within the physical pipeline. For example, if a table log has $e_{log} = 5000$ entries, the number of entries assigned to that logical table should be at least 5000. Hence:

$$\forall log : \sum_{mem,st} WordsAssigned_{st,log,mem} \geq e_{log}. \tag{1}$$

Capacity Constraint: Recall that action memory and statistics memory are captured as extra memory block overhead per stage. For each memory type, aggregate memory assignment for logical tables to a stage must not exceed physical capacity of that stage:

$$\forall mem, st : \sum_{log} MemoryUsage_{st,log,mem} \leq B_{st,mem}. \tag{2}$$

Dependency Constraint: The solution must also respect dependencies between logical tables. If table log_2 depends on log_1's results, the earliest possible stage containing log_2's entries must wait until the results of Table log_1 are known, which could occur at log_1's last stage, since tables are allowed to span multiple pipeline stages.

$$\begin{aligned} &\forall log_1, log_2 : \\ &\qquad \text{If} \quad Dependency_{log_1,log_2} > 0 : \\ &\qquad\qquad End_{log_1} \leq Start_{log_2}. \end{aligned} \tag{3}$$

If log_1 must completely finish executing before log_2 begins (e.g., match dependencies), then the inequality in Equation 3 becomes strict.

Objective Functions Greedy methods often cannot provide an optimal solution. Objective functions are a key advantages of ILP. For example, to minimize the number of stages we minimize:

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

where for all stages st:

$$\text{If} \sum_{mem,log} MemoryUsage_{st,log,mem} > 0 :$$

$$maxStage \geq st.$$

On the other hand, consider minimizing total pipeline latency for say RMT. Building on Equation 3's characterization of dependencies, we calculate the latency of a particular stage st based on the dependencies of tables assigned in stage st. We assign all stages st a start time, $Time_{st}$, where $Time_{st}$ is strictly increasing with st. Note that the end times are irrelevant, as all stages take 12 cycles to complete. Then if log_2 has a match dependency (12 cycle wait) on log_1, where the last stage of log_1's assignment and the first stage of log_2's assignment are stages End_{log_1} and $Start_{log_2}$, respectively, we can constrain the start time of $Start_{log_2}$ as follows:

$$Time_{End_{log_1}} + 12 \leq Time_{Start_{log_2}}.$$

Similar constraints can be constructed for action dependencies (3 cycle wait); there are no latency constraints for successor and reverse-match dependencies. Then we minimize the start time of the last stage, stage M:

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

We focus on minimizing either stages or latency for the remainder of this paper. Next, we describe our ILP approaches for FlexPipe and RMT. Our discussion is based on Figure 2, which separates our program into a switch-specific preprocessor and a switch-independent compiler. Besides the common assignment, capacity, and dependency constraints, there are additional switch-specific constraints.

5.2 Large granularity approach

In our first approach we use for RMT, the smallest unit of measurement are the packing units suggested in Section 3.

Basic Assignment: For a given logical table log, recall that packing unit $p \in \{1, \ldots, p_{max}\}$ is the memory arrangement that yields a wider match entry of p words packed together. The user can specify p_{max} is the maximum packing unit factor available. The solver's assignment of logical table log to stage st's memory type mem determines the number of packing units p for $p \in \{1, \ldots, p_{max}\}$, denoted as $PackingUnits_{st,log,mem,p}$. We support multiple packing units in a stage for a single logical table.

Layouts for Packing Units: The preprocessor computes $BlocksOccupied_{log,mem,p}$, the number of memory blocks required to match on a packed word entry. The number of blocks is a function of the width of the the logical match word, the number of match words p, and the width w_{mem} of a type mem memory block. The ILP solver computes the match memory blocks and words assigned, denoted $MatchBlocks_{st,log,mem}$ and $WordsAssigned_{st,log,mem}$ respectively, for an assignment of packing units in stage *st* and memory type *mem* as follows:

$$MatchBlocks_{st,log,mem} = \sum_{p=1}^{p_{max}} PackingUnits_{st,log,mem,p} BlocksOccupied_{log,mem,p}$$

$$WordsAssigned_{st,log,mem} = \sum_{p}^{p_{max}} PackingUnits_{st,log,mem,p}(p \cdot d_{mem}),$$

where d_{mem} is the maximum number of entries that can fit a single memory block of type *mem*.

With this packing unit definition, we can implement the assignment, capacity, and dependency constraints.

Assignment and Capacity Constraints: Using the variable $WordsAssigned_{st,log,mem}$, we define the general assignment constraint as in Equation 1. In order to compute the capacity constraint (Equation 2) from $MatchBlocks_{st,log,mem}$, we must reference the preprocessor. Define the assignment overhead of assigning table *log* in memory type *mem* as an overhead factor $Overhead_{mem,log}$. For every match block assigned, an additional $Overhead_{mem,log}$ blocks are required. The total amount of memory that table *log* occupies in stage *st* becomes:

$$MemoryUsage_{st,log,mem} = MatchBlocks_{st,log,mem}(1 + Overhead_{mem,log})$$

Dependency Constraints: To implement the dependency constraint of Equation 3, we compute the starting and ending stages of a table *log*, denoted $Start_{log}$ and End_{log}. Note that $Start_{log}$ can be further defined as the minimum stage that contains a non-zero $PackingUnits_{st,log,mem,pf}$ assignment; End_{log} can be defined similarly as the maximum stage with non-zero assignment.

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. Thus it is trivial to compute the number of subunits a particular stage *st* occupies based on the $PackingUnits_{st,log,mem,pf}$ assignments for this stage.

Pipeline Latency Objective: In RMT, instead of squeezing all logical tables into a small number of memory blocks, spreading the tables across stages can improve pipeline latency, as we will see later. Pipeline latency in RMT depends only on the start times of each stage, denoted $Time_{st}$.

Scaling: The number of variables scales as $O(MNKp_{max})$, where N is the number of tables, M is the total number of stages, K is the number of memory types, and p_{max} is the maximum packing unit the user specifies. In general, this can scale as $O(MN)$, since p_{max} and K are generally small.

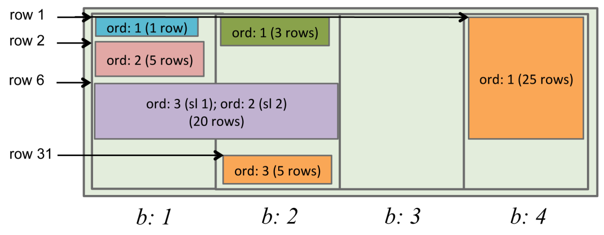


Figure 7: Valid assignment of tables in memory blocks of one stage in FlexPipe. The purple table occupies the first two memory blocks, but different sets of tables share the first two memory blocks.

5.3 Small granularity approach

Since the first approach does not appropriately model the memory block sharing feature of FlexPipe, our second approach uses variables of a fine granularity.

To support configurations as in Figure 7, the position of each memory block in each stage is important, as the blocks associated with a longer word assignment, such as the violet table, will contain different sets of assignments. Note that this issue does not arise in RMT; all memory blocks that contain a logical table will be uniform, and solution can be rearranged to group together all memory blocks of a particular table assignment in a stage. We thus index the memory blocks as $b \in 1, \ldots, B_{st,mem}$, where $B_{st,mem}$ is the maximum number of *mem* type blocks in stage *st*.

Basic Assignment: A valid solver should determine the number of entries of a logical table to assign to a stage. Note that the exact entry in a memory block at which the solver begins is important; if a long match entry like IPv6 is assigned row 20 of a 36 bit wide prefix match block, it will overflow into the adjacent block, where it must also be assigned row 20. 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_{mem} are the first and last rows, respectively, of a block.

Thus we define two variables for each assignment of table *log* to the *b*th block of memory type *mem* in stage *st*: the starting row $LogStartRow_{log,mem,b}$ and the number of consecutive rows $StartRowsAssigned_{log,mem,b}$. Note that if a second table, log_1, has entries in an adjacent block b_1, but the entries are wide and overflow into block *b*, $StartRowsAssigned_{log_1,mem,b} = 0$ becase the starting row for log_1 was not assigned in this block; similarly, $LogStartRow_{log,mem,b}$ becomes irrelevant.

Layouts for Row Assignments: We must constrain the assignment of multiple tables to a block b (in type mem) by ensuring that the assigned rows do not overlap with each other within the block. In order to implement our constraint, we also introduce the notion of order, as shown in Figure 7, where the dark red assignment has order $ord = 1$, because it has the earliest row assignment. We denote this concept of order in the variable $ord \in \{1, \ldots, ord_{max}\}$, where ord_{max} is the maximum number of logical tables that can share one given memory block.

In this sense, we simply need to constrain assignment within a block b by ensuring that the number of rows of the logical table with a previous order does not overlap into the current order. Define two additional variables, $StartRowOrd_{mem,b,ord}$ and $OrdRows_{mem,b,ord}$, the start row and quantity of rows of table with order ord. These two variables can be computed from $LogStartRow_{log,mem,b_1}$ and $StartRowsAssigned_{log,mem,b_1}$, where b_1 could be an earlier block. If ord-th entries are assigned:

$$StartRowOrd_{mem,b,ord-1} + OrdRows_{mem,b,ord} \leq \leq StartRowOrd_{mem,b,ord}$$

Assignment Constraint: Recall that a block b that is associated with table log's assignment will either contain the beginning of log's words or a later portion of the words. Thus it is sufficient to calculate the total number of words assigned for table log in a stage st based on the start rows assigned, in order to implement the assignment constraint (Equation 1). Our FlexPipe model does not contain action or statistics memory overhead:

$$WordAssigned_{st,log,mem} = \sum_{b=1}^{b_{st,mem}} StartRowsAssigned_{log,mem,b}$$

Capacity Constraint: Simply put, the capacity constraint of Equation 2 restricts the number of rows we can assign in a memory block. Thus it is sufficient to check to constrain the last order, ord_{max}:

$$StartRowOrd_{mem,b,ord_{max}} + OrdRows_{mem,b,ord_{max}} \leq d_{mem}$$

Dependency Constraints: Fortunately, the dependency analysis is similar to that of RMT in Section 5.2, 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 instead minimize the total number of blocks used across all stages. *Scaling:* The number of variables scalre as $O(ord_{max}NB_{all})$, where B_{all} is the total number of memory blocks of all memory types across all stages. This is reasonable for a small chip like FlexPipe which has relatively few memory blocks of each type, but it scales poorly with RMT, which has over 1000 memory blocks.

Name	Switch	N Tables	Dependencies		
			Match	Action	Other
L2L3 _simple	RMT	16	4	0	15
	FlexPipe	13	12	0	4
L2L3 _mTag	RMT	19	6	1	16
L2L3 _complex	RMT	24	23	2	10

Table 4: Logical program benchmarks to run on RMT and flexPipe.

6 Experimental Results

6.1 Setup

Switch Specifications: We use the RMT and FlexPipe configurations shown in Table 3. Furthermore, our RMT model also has input crossbars for SRAM, TCAM, and action input for each stage, as described in Section 2. FlexPipe has no additional resource constraints.

Benchmarks: We use two sets of benchmarks each for RMT and FLexPipe, each gathered from the vendor. Since the FlexPipe chip has only five stages with few memory blocks per stage, we use two simple benchmarks: a simple L2/L3 program similar to Figure 3, and a similar program with support for the mTag toy example mentioned in [1].

RMT can support larger logical table sizes with more intricate dependencies. We use three benchmarks: an expanded version of the FlexPipe simple L2/L3 program, which includes additional small tables to dictate control flow; an expanded version of the simple L2/L3 program with mTag support; and a more complex L2/L3 program that creates and operates on metadata and includes more ACLs and features like URPF and ECMP.

A summary of the logical features of these FlexPipe and RMT benchmarks that are important to our solution are in Table 4.

ILP Objectives: Our ILP experiments for ILP minimize various objectives such as Pipeline Latency, weighted number of memory blocks used, maximum stage used etc. For Flexpipe, ILP minimizes the last block assigned for each memory type, which is simply one way of fitting the tables in the pipeline.

Experiments: We run each experiment setup on our FFD, FFL, and ILP solvers and compare the the three approaches by analyzing the number of stages a solution occupies, the pipeline latency, and the solver runtime. In all experiments, our ILP optimizes for minimum pipeline latency.

1. L2L3_simple: We use ten different versions of the L2L3_simple, each varying table sizes, by randomly generating over 100 sets of table entries for the main tables (e.g., routing, switching, and ACL tables) We choose numbers form ranges that would be realistic for a small datacenter or enterprise network. We then select programs where at least one of the greedy heuristics has a solution that fits in 16 stages. We then compare how FFL, FFD, and ILP fit ten of these programs in such a 16-stage chip, then evaluate the approaches on a slightly smaller 15-stage chip.

2. L2L3_complex: We tested this benchmark with one set of table sizes and widths characteristic of small datacenters within an enterprise or small network. From this logical program benchmark, we evaluate how FFL, FFD, and ILP fit on switch architectures, where the total number of stages is in the range $M \in \{15,\ldots,21,32\}$.

For FlexPipe, our main criterion is whether the approach provides a feasible solution that fits all tables in the switch. As a result we generate different version of the L2L3_simple program and select those programs where the greedy heuristic manages to fit all but one logical table. We then assess whether ILP can perform better and find a feasible solution for these programs in the FlexPipe switch.

6.2 Results

The results for running RMT are shown in Table 5a. For FlexPipe, we generated 95 different versions of the L2L3_simple program, where for each version the tables sizes should be large enough to take up at least one tenth of the switch's total memory. We then tried to fit these programs in the FlexPipe switch using the greedy heuristic. The greedy heuristic was able to fit 86 of the 95 versions successfully. The ones that it did not fit were either infeasible or could possibly be fitted using ILP (TODO: Run these completion using ILP solver.)

6.3 Analysis: ILP versus Greedy

There are many realistic cases where ILP needs fewer stages to fit a program than greedy. This is important when switches have limited stages. Even in cases when there are enough stages for greedy to fit a program, Greedy can perform 25% worse than ILP in terms of objectives such as pipeline latency. Three common situation when ILP outperforms greedy are when there are *multiple conflicting metrics*, *multiple memory types* and *complicated objectives*. We examine each in turn.

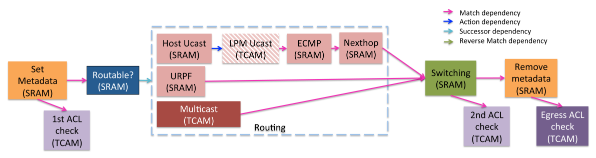


Figure 9: Dependency graph for L2L3_complex.

1. Conflicting Metrics: Greedy heuristics typically look at one metric when assigning tables e.g., the resources used or the number of match dependencies in the control flows following the table. These metrics can sometimes conflict with each other, e.g., there may be a small table at the beginning of a long control flow, which a resource based heuristic could overlook and start assigning only when it's too late.

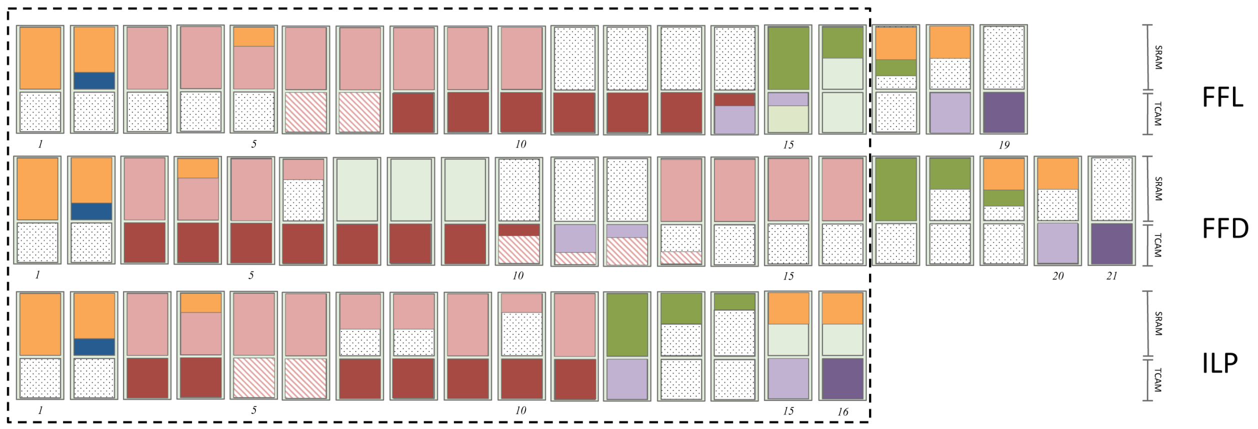
We have observed such examples in our benchmarks-we tried to fit L2L3_complex using FFD in a 16 stage chip (see Figure 8a). FFD picks a 2 subunit wide multicast table (the dark red table in the figure) before a 1 subunit wide unicast LPM table (the pink striped table in the figure), both of which need to go in TCAM. However, the unicast prefix table has more match dependencies following it, than the multicast table (see Figure 9). By the time the unicast table is completely assigned to TCAM in stage 13, there aren't enough stages left for the its dependent tables.

We can also provide synthetic examples with just one memory type, where neither heuristic can fit the table, because the best approach needs to look at multiple metrics and not just one (see Figure 10).

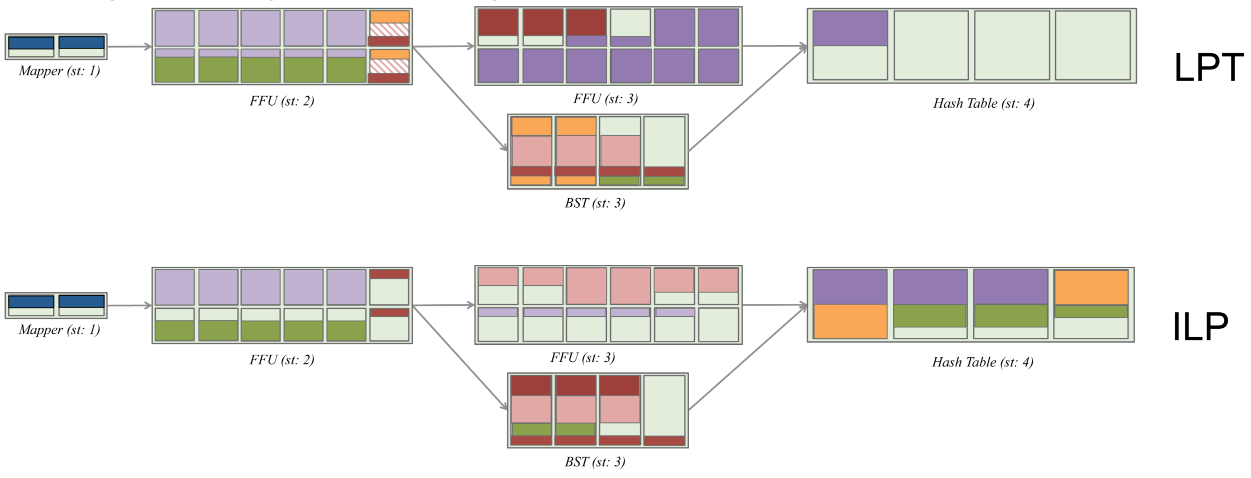
There can be problems with conflicting metrics in FlexPipe too, even though tables are sorted in order of how constrained they are in terms of both dependencies and allowed memory types. TODO(): Describe counterexample with pictures.

2. Multiple memory types Our greedy heuristics assign a table greedily in the available space i.e., they fill up a stage as far as possible before moving to the next stage. This may not be the right strategy when there is more than one type of memory.

The first reason is that prefix tables that use TCAM may still need some SRAM blocks for their action memory. For an exact match table picked before some prefix table, the greedy heuristics use up SRAM blocks in one or more stages until the table is completely assigned. However this renders the TCAM blocks unusable for prefix tables that have action data. On the other hand, leav-



(a) Sample FFD, FFL, and ILP solutions for L2L3_complex for and RMT chip with 16 stages. The greedy heuristics need more than 16 stages, the extra stages are also shown in figure.



(b) Sample LPT and ILP solutions for L2L3_simple, FlexPipe chip.

Figure 8: Sample greedy and ILP solutions.

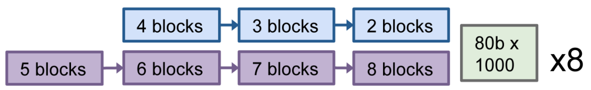
Program Name		*M* Stages	FFD Stages Used	FFD Latency (cycles)	FFD Exec. Time	FFL Stages Used	FFL Latency (cycles)	FFL Exec. Time	ILP Stages Used	ILP Latency (cycles)	ILP Exec. Time
L2L3 simple		15	null	null	0.11 s	null	null	0.27 s	15	39	71 s
	min lat.	16	16	45	0.13 s	16	47	0.25 s	16	39	70 s
	max lat.		null	null	0.11 s	16	48	0.28 s	16	42	84 s
L2L3_mtag		15	11	58	0.33 s	15	58	0.47 s	15	49	660 s
		16	11	59	0.32 s	15	59	0.48 s	16	49	310 s
L2L3_complex		16	null	null	0.69 s	null	null	0.67 s	16	106	6300 s
		19	null	null	0.76 s	19	118	0.8 s	19	104	1400 s
		21	21	135	0.8 s	19	120	0.89 s	21	104	6100 s
		32	21	146	0.86 s	19	131	0.86 s	32	104	3100 s

(a) Benchmark results for RMT.

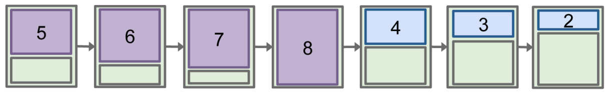
Program Name		*M* Stages	LPT Blocks Used	LPT Exec. Time	ILP Blocks Used	ILP Exec. Time
L2L3 simple	min size.	5	31	51 s	31	100 s
	max size.		null	51 s	34	12245 s

(b) Benchmark results for FlexPipe.

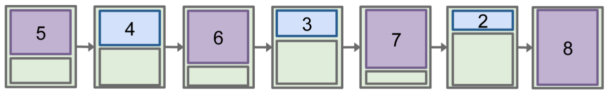
Table 5: Experiment results. A null indicates that the solver cannot fit the program in the given switch.



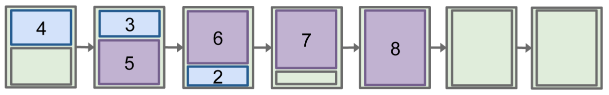
(a) Greedy Counterexample setup. Blue and purple DAGs are logical tables with match dependencies; each table occupies a different number of blocks at packing unit 1. Each stage holds 8 memory blocks with dimension 80b ×1000.



(b) Greedy Counterexample for RMT- FFD solution (7 stages). The wider purple DAG is assigned first.



(c) Greedy Counterexample for RMT- FFL solution (7 stages). Tables cannot share stages.



(d) Greedy Counterexample for RMT- Optimal solution (5 stages). By assigning the blue DAG first, the remaining tables can share stages.

Figure 10: A toy RMT example where greedy performs much worse than ILP.

ing some room for action memory in SRAM in every stage helps later prefix tables used the switch TCAM more efficiently.

We have observed examples in our benchmarks- we tried to fit L2L3_complex using FFL in a 16 stage chip (see Figure 8a). FFL can start as early as Stage 3 when it starts to assign the multicast prefix (dark red in figure) based on dependencies (it must start after the blue 'Routable' table end). However it is forced to start tables later than needed in stage 8, because it didn't leave any SRAM blocks for the action memory in stages 3 through 5. As a result, tables that depend on the multicast tables must start later too, in stage 15.

We modified our heuristic to reserve a fixed number of RAMs for action memory for ternary tables in each stage. Though this fixes the above problem in many cases, it is hard to predict exactly how many RAMs must be reserved in each stage. Reserving more than required can waste RAMs and end up using more stages than needed.

TODO(): Describe FlexPipe counter-example too.

3. Complicated objectives It is easy to design heuristics that only fit tables in the switch chip based on bin-packing or job scheduling. But users may want to optimize for more complicated objectives such as pipeline latency or a weighted cost of total memory used.

It can be difficult to come up with a heuristic to minimize pipeline latency- There are two conflicting goals. A solver needs to space out tables with dependencies to reduce pipeline latency. But it also needs to pack tables tightly enough so that they fit the switch. It seems hard to find the right balance using a simple greedy heuristic.

Our greedy heuristics focus on using the minimum number of stages so that the tables can at least fit the switch.

For a given program, a greedy heuristic always uses the same number of stages. This could be more stages than optimal. More alarmingly, it could be more than the number of available stages as we saw in the 16-stage L2L3_complex example (Table 5a). On the other hand an ILP that specifically optimizes for pipeline latency, uses all the available stages and no more, to space out the tables appropriately and minimize the pipeline latency. For example, in the 32-stage L2L3_complex example (Table 5a), the ILP uses all 32 stages and saves up to 32 cycles compared to the better heuristic.

We conclude that new objectives like pipeline latency are easy to implement via ILP but might require a completely new approach when it comes to heuristics.

7 Conclusion

We defined the problem of mapping logical tables in packet processing programs to two very different switch architectures. We introduced a set of abstractions such as memory types, topological order of stages and packing units to hide the idiosyncracies of different switches. We explored two mapping approaches- one based on greedy heuristics and another based on a modular ILP. Our modular approach allows us to be more flexible and develop faster.

Our ILP approaches for the two chips share the same fundamental constraints. However because the two chips have different granularities for assigning logical tables to switch memory, we used a different set of basic assignment variables for each chip, to implement the fundamental constraints.

We evaluated our greedy heuristics and the ILP approach on realistic benchmarks. While fitting the tables to a switch is the basic criterion, we also compared how well our solvers in minimizing pipeline latency on the long RMT pipeline. We found that for RMT, there are realistic configurations where greedy approaches can fail to fit the program, and need up to 27% more resources on the same benchmark. Three common 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 rather than ad hoc and inefficient greedy solutions.

References

[1] 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).

[2] 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.

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

[4] Chaitin, G. J. Register allocation & spilling via graph coloring. In *ACM Sigplan Notices* (1982), vol. 6, ACM, pp. 98–105.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] XILINX, INC. *DSP: Designing for Optimal Results*. Xilinx, Inc., 2005.