dRMT: Disaggregated Programmable Switching

Sharad Chole[1], Andy Fingerhut[1], Sha Ma[1], Anirudh Sivaraman[2], Shay Vargaftik[3], Alon Berger[3], Gal Mendelson[3], Mohammad Alizadeh[2], Shang-Tse Chuang[1], Isaac Keslassy[3,4], Ariel Orda[3], Tom Edsall[1]
[1] *Cisco Systems, Inc.* [2] *MIT* [3] *Technion* [4] *VMware, Inc.*

ABSTRACT

We present dRMT (disaggregated Reconfigurable Match-Action Table), a new architecture for programmable switches. dRMT overcomes two important restrictions of RMT, the predominant pipeline-based architecture for programmable switches: (1) table memory is local to an RMT pipeline stage, implying that memory not used by one stage cannot be reclaimed by another, and (2) RMT is hard-wired to always sequentially execute matches followed by actions as packets traverse pipeline stages. We show that these restrictions make it difficult to execute programs efficiently on RMT.

dRMT resolves both issues by disaggregating the memory and compute resources of a programmable switch. Specifically, dRMT moves table memories out of pipeline stages and into a centralized pool that is accessible through a crossbar. In addition, dRMT replaces RMT's pipeline stages with a cluster of processors that can execute match and action operations in any order.

We show how to schedule a P4 program on dRMT at compile time to guarantee deterministic throughput and latency. We also present a hardware design for dRMT and analyze its feasibility and chip area. Our results show that dRMT can run programs at line rate with fewer processors compared to RMT, and avoids performance cliffs when there are not enough processors to run a program at line rate. dRMT's hardware design incurs a modest increase in chip area relative to RMT, mainly due to the crossbar.

CCS CONCEPTS

• **Networks** → **Programmable networks**; *Routers*;

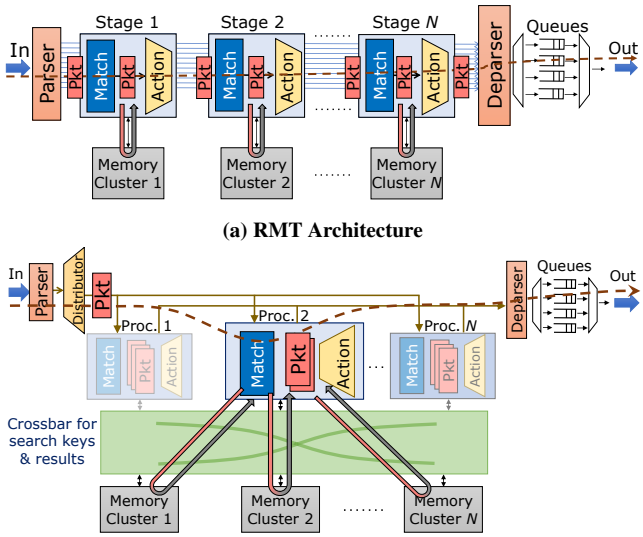
KEYWORDS

Programmable switching; packet processing; RMT; disagreggation

ACM Reference format:
Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098823

SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098823



(a) RMT Architecture

(b) Disaggregated RMT (dRMT) Architecture

Figure 1: Comparison of the RMT [16] and dRMT architectures. dRMT replaces RMT's pipeline stages with run-to-completion match-action processors, and separates the memory clusters from the processors via a crossbar. The dashed arrows show the flow of a packet through each architecture.

1 INTRODUCTION

Historically, high-speed packet switching chips have been architected as a pipeline of match-action stages. For each incoming packet, each stage (1) extracts specific packet header bits to generate a match key, then (2) looks up this key in a match-action table, and finally (3) uses the match result to run an action. For instance, a stage could extract the packet's IP destination address, look up this IP address in a forwarding table, and use the result to determine the outgoing port. In recent years, programmable switches [2, 7, 14, 16] have emerged, allowing a switch pipeline's match-action stages to be programmed in languages like P4 [15].

RMT. The predominant architecture for programmable switches is the Reconfigurable Match-Action Table (RMT) architecture [16]. As illustrated in Figure 1a, RMT uses a pipeline of match-action stages, similar to conventional fixed-function switches. However, RMT makes the match-action stages programmable; programmers can specify the set of headers to match on, the type of match to perform (exact, ternary, *etc.*), and compose their own compound actions out of primitive actions.

RMT's pipeline stages contain three kinds of hardware resources: (1) match units that extract the header bits to form match keys, (2) table memory in a local memory cluster, and (3) action units that

programmatically modify packet fields. For example, a stage might have a match unit to extract up to 8 80-bit keys from the packet header, 11 Mbit of SRAM and 1.25 Mbit of TCAM for tables, and an action unit to modify up to 32 packet fields in parallel.

By connecting these resources in a sequential pipeline, RMT simplifies wiring significantly [16]. However, RMT suffers from two key drawbacks as a result of its pipelined architecture. First, because each pipeline stage can only access local memory, RMT must allocate memory for a table in the same stage that extracts its match key and performs its action. This conflates memory allocation with match/action processing, which makes table placement challenging [22] and can result in poor resource utilization (§2.1). For instance, when a large table does not fit in one stage, it has to be spread over multiple stages. But, in the process, the match/action units of (all but one of) these stages are wasted, unless there are other tables that can execute in parallel.

Second, RMT's hard-wired pipeline can only execute operations in a fixed order: a match followed by an action in stage 1, then a match followed by an action in stage 2, and so on. This rigidity can lead to under-utilization of hardware resources for programs where matches and actions are *imbalanced*. For example, a program with a default action that does not need a preceding match [12], such as decreasing the packet's TTL, wastes the match unit and table memory in the pipeline stage that runs the default action. Moreover, since packets can only traverse the pipeline sequentially, a program that does not fit in the available hardware stages must *recirculate* packets through the pipeline; this cuts throughput in half, even if the program needs only one extra processing stage (§2.2).

dRMT. We propose dRMT (disaggregated RMT), a new architecture for programmable switches that solves both problems confronting RMT. dRMT's key insight is to disaggregate the hardware resources of a programmable switch. As illustrated in Figure 1b, dRMT disaggregates:

(1) **Memory:** dRMT separates the memory for tables from the processing stages and makes them accessible via a crossbar. The crossbar carries the search keys and results back and forth between the match/action units and memories.
(2) **Compute:** dRMT replaces RMT's sequentially-wired pipeline stages with a set of *match-action processors*. Match-action processors consist of match and action units, similar to RMT's pipeline stages. But, unlike pipeline stages, packets do not move between dRMT processors. Instead, each packet is sent to *one* dRMT processor according to a round-robin schedule. The packet resides at that processor, which runs the entire program for that packet to completion.

Memory and compute disaggregation provide significant *flexibility* to dRMT. First, memory disaggregation decouples the memory allocation for a table from the hardware that performs its match-action processing. Second, compute disaggregation makes it possible to interleave match/action operations in any order at a processor, both for a given packet and across different packets. Finally, compute disaggregation allows for *inter-packet concurrency*, the ability for a processor to perform match/action operations on more than one packet at a time. This flexibility results in increased hardware utilization for dRMT relative to RMT, reducing the amount of hardware (*e.g.,* number of stages/processors) necessary to run a program at line rate. Equivalently, it increases the set of programs that a fixed amount of hardware can execute at line rate.

dRMT's run-to-completion packet processing model has been previously used in some network processors [4, 10, 11]. However, these network processors do not guarantee deterministic packet throughput and latency. Nondeterminism occurs in network processors for a variety of reasons, including cache misses and contention in the processor-memory interconnect. In dRMT, we show how to schedule the entire system (processors and memory) at compile time such that no contention ever occurs. Given a P4 program, our scheduling algorithm calculates a static schedule at compile time, guaranteeing a deterministic throughput and latency (§3).

We evaluate dRMT using four benchmark P4 programs (§4), three derived from the open-source switch.p4 [13] program and another proprietary program from a large switching ASIC manufacturer. Across these programs, we find that dRMT requires 4.5%, 16%, 41%, and 50% fewer processors than RMT to achieve line-rate throughput (1 packet per cycle). We also find that dRMT reduces the number of processors required for line-rate throughput by an average of 10% (up to 30%) on 100 randomly generated programs with characteristics similar to switch.p4. Further, dRMT's throughput degrades gracefully with fewer processors, while RMT's performance falls off a cliff if the program needs more stages than provided by the hardware.

We present a hardware design for dRMT (§5) and analyze its feasibility and chip area cost (§6). dRMT's flexibility relative to RMT comes at some additional chip area cost to (1) implement a crossbar that is absent in RMT, and (2) implement a match-action processor that stores and executes an entire P4 program, unlike an RMT stage that only stores and executes a fragment of the P4 program. We present architectural optimizations that trade off modest restrictions for a lower cost. While we have not built a dRMT chip, our analysis shows that it is possible to implement dRMT with a chip area comparable to RMT for the same number of processors/stages. For example, a dRMT chip with 32 processors costs about 5 mm^2 more area than RMT with 32 stages, a modest increase relative to the total chip area of a typical switching chip (>200 mm^2) [19]. dRMT's scalability is limited by the wiring complexity of the crossbar. Scaling the crossbar far beyond 32 processors, which already requires careful manual place and route, may be difficult. Fortunately, switching chips are unlikely to need more than 32 processors (*e.g.,* a state-of-the-art programmable switch has 12 stages [1]).

The dRMT project page [5] contains the code required to reproduce our experimental results. It also contains an extended paper with proofs of all theorems described in this paper.

2 THE CASE FOR DISAGGREGATION

2.1 Memory disaggregation

In RMT, each pipeline stage can only access its local memory cluster. As a result, a table must reside in the memory of the same stage that extracts its search key and executes its action.[1] This leads to a coupling between two problems: (1) choosing which match and action

[1] In theory, the action for a table could be deferred to a stage *after* the lookup. But this requires passing the results of the lookup between stages along with the packet, consuming extra space in the packet header vector [16]. RMT compilers [22] typically avoid deferring actions for this reason.

operations are executed by each stage, and (2) placing the program's tables into memory clusters. The first problem involves scheduling match and action operations across stages such that the program's dependencies are not violated (see §3 for details). The second problem concerns packing the tables into the memory clusters. dRMT decouples these two problems by enabling all processors to access all memory clusters via a crossbar. This has several advantages.

Improving hardware utilization. The most important benefit of memory disaggregation is significantly increased flexibility for mapping operations and tables to hardware resources more efficiently.

Example 2.1 (Parallel searches). Consider a program with four tables whose searches can be done in parallel. If the search keys for the four tables can be extracted in one RMT stage, but their total table size exceeds the capacity of one memory cluster, then some of the tables must be moved to later stages. Similarly, if the tables fit in one memory cluster, but the key width exceeds the key extraction capacity of one stage, some tables must be moved to later stages. In either case, some resources—key extraction hardware or table memory—are left unused in the first stage. With dRMT, however, key extraction and table placement are decoupled; a processor can extract the four search keys and send them over the crossbar to whichever memory clusters store the tables. (See §5.3 for the crossbar's design details.)

Example 2.2 (Large table). A large table may not fit entirely within one memory cluster and may need to be split across multiple stages. With RMT, each stage must search its part of the table—extracting the same key multiple times—and the partial match results must be combined together. The table's action cannot be performed until the last of the stages, potentially wasting the action units in all but the last stage. In dRMT, the crossbar can multicast the search key to multiple memory clusters, where the partial searches would be done simultaneously. With a small amount of result-combining logic in the return path from the memory clusters back to the processors, the processor will only receive the highest-priority result.

Independently scaling processing/memory capacity. Memory disaggregation makes it straightforward to select the number of processors and the number of memory clusters independent of each other, based on the kinds of programs one wishes to execute. For example, a designer can trivially add a TCAM memory cluster that is accessible via the crossbar to increase TCAM capacity. By multicasting search keys, the new TCAM can be allocated to any table that needs it. By contrast, increasing the memory allocated to an RMT stage without increasing its match/action capacity risks under-utilizing the memory.

Simpler compilation. An RMT compiler must place tables across pipeline stages while respecting the dependencies between program operations [22]. A dRMT compiler needs to solve two simpler problems: (1) packing tables into memory clusters, which can be solved using simple bin packing, and (2) scheduling operations on processors. An important property of dRMT is that these two problems are *decoupled*; tables can be placed in memory clusters irrespective of how operations are scheduled, and vice-versa. This makes compiling programs to dRMT conceptually simpler than RMT. We discuss the dRMT scheduling problem in detail in §3.

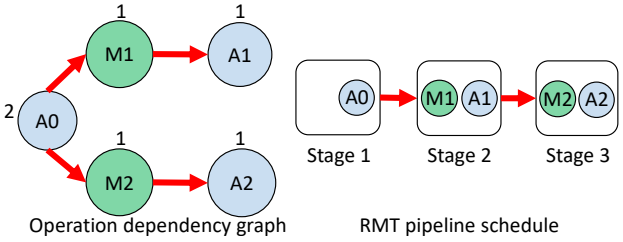


Figure 2: A toy program and its schedule on an RMT pipeline.

proc. \ cycle	0	1	2	3	4	5	6
0	A_0	M_1	M_2	A_1&A_2			
1		A_0	M_1	M_2	A_1&A_2		
0			A_0	M_1	M_2	A_1&A_2	
1				A_0	M_1	M_2	A_1&A_2

Figure 3: Schedule for toy program on dRMT with 2 processors.

The benefits of memory disaggregation are not limited to dRMT. One could similarly add a crossbar to RMT, connecting all pipeline stages to all memory clusters, and providing the same benefits. However, as we discuss next, dRMT takes disaggregation a step further by getting rid of the pipeline entirely and disaggregating the *compute resources* as well. Our results show that compute disaggregation is essential to achieving the full potential of disaggregation (§4).

2.2 Compute disaggregation

The RMT architecture enforces a rigid match-then-action sequence of operations in the pipeline. In dRMT, we allow matches and actions to be interleaved in any order on a processor, as long as dependencies and resource constraints are respected. This has several benefits.

Improving hardware utilization. Compute disaggregation further increases flexibility to order operations in a way that maximizes hardware utilization. We demonstrate this advantage using a toy program whose dependencies are given by the directed acyclic graph (DAG) in Figure 2. We assume that every edge mandates a minimum latency of one cycle between the operations on the edge; the numbers on the nodes represent their (match or action) resource requirements. We schedule this DAG to run both on RMT and dRMT, assuming both can perform up to 1 match every clock cycle and 2 actions every clock cycle in each stage/processor.

In the RMT pipeline, this DAG requires at least 3 stages because there is insufficient match capacity to run matches M1 and M2 in parallel, and both M1 and M2 have to follow A0 (Figure 2). The problem with this schedule is that the match unit in the first stage is stranded. dRMT can schedule the same program using just 2 processors, as shown in Figure 3. Each row shows the sequence of operations for one packet, executed on one processor. Notice that packets are sent to processors in round-robin order: the packet that arrives in cycle k runs on processor $k \bmod 2$. Also notice that the operations in each processor do not exceed the processor's capacity of 1 match and 2 actions per clock cycle.

Eliminating performance cliffs. Packet switching ASICs typically have a recirculation path, by which packets that do not finish within a single pass can be sent back to the beginning (mingled with newly arriving packets). In a pipelined architecture, if a particular program cannot be scheduled to fit within the available match-action stages, one may split the program into multiple passes. The packet rate for a K-pass schedule is $1/K$ of the system's maximum rate.

By contrast, with dRMT, throughput degrades *gracefully* as program complexity increases. For example, if a dRMT system can support M table searches at a rate of 1 packet per clock cycle, it is possible to support a program that requires $M' > M$ table searches at a rate close to M/M' packets per clock cycle. This simply requires increasing the time that packets spend in their processors, and reducing the rate at which packets are sent to processors.

3 SCHEDULING FOR DETERMINISTIC PERFORMANCE

We now describe the dRMT scheduling problem and develop a scheduling technique to execute a P4 program efficiently on dRMT hardware while providing *deterministic* throughput and latency. Specifically, given a P4 program, we seek a fixed schedule—precomputed at compile time—that satisfies two constraints: (1) the P4-program-specific dependencies, and (2) the dRMT architecture's resource constraints such as the match and action capacities of each processor. An important aspect of our formulation is that the same schedule is repeated across processors in round-robin fashion. This simplifies the problem significantly, as it greatly reduces the space of schedules that we must search over. It also gives rise to a unique set of cyclic constraints that our schedules must satisfy.

We begin by describing the dRMT scheduling problem, and then introduce our main theoretical results. In particular, we show the NP-hardness of fixed scheduling for deterministic performance, and then formulate the problem as an Integer Linear Program (ILP).

3.1 Scheduling problem

First, we present the P4 program dependency constraints and the dRMT architectural constraints. Then, we introduce the scheduling optimization problem given these two types of constraints, and also present an illustrative example after the definitions.

Program dependencies. Each packet entering dRMT follows the control flow [22] dictated by a P4 program. This control flow specifies how the packet headers are to be processed. To express a program's dependencies, we define an *Operation Dependency Graph (ODG)*, *i.e.,* a directed acyclic graph (DAG) in which the nodes represent the match and action operations executed by a packet while traversing the switch, and the edges describe the dependencies between these operations.

An edge between two nodes dictates that any valid schedule must perform the first before the second. The edge is annotated with a latency that specifies the minimum time separation between the operations. For an edge from node A to B, this latency is the time that it takes to complete the operation on node A. We assume that a table match takes ΔM clock cycles, and an action takes ΔA clock cycles, meaning that an operation dependent on a match or an action has to wait ΔM or ΔA cycles respectively.

For operations that are conditionally executed (*e.g.,* based on a predicate or whether there was a table hit or miss), we conservatively assume that both branches of the condition are executed and schedule both, even though only one will execute at run-time. This is equivalent to executing all action nodes within a conditional branch speculatively, and committing any side effects based on the conditional test afterwards, similar to the speculative execution model adopted by RMT [16]. This worst-case assumption simplifies the scheduling problem. In practice, we find that it still allows us to schedule programs efficiently (§4).

Example 3.1 (P4 program). Figure 4a depicts the control flow of a simple P4 program that supports unicast and multicast routing (it is a fragment from the L2L3 program in [22]). Figure 4b depicts the corresponding ODG. M_0, M_1, M_2 and M_3 can be executed concurrently. Action A_1 must precede A_2 because they both write to the same fields, and A_2's outcome must be the end result.

The ODG is similar to the Table Dependency Graph (TDG) [22], proposed by Jose *et al.* to represent P4 program dependencies. But the ODG is simpler. The TDG annotates edges based on the type of dependency (*e.g.,* match, action, and reverse-match [22]). By splitting each table in the TDG into two distinct match and action nodes (with an edge between them), the ODG lets us represent all these dependencies in a unified way using appropriate edge latencies.

Architectural constraints. A schedule must respect several dRMT architectural constraints. We detail these constraints before providing an illustrative example.

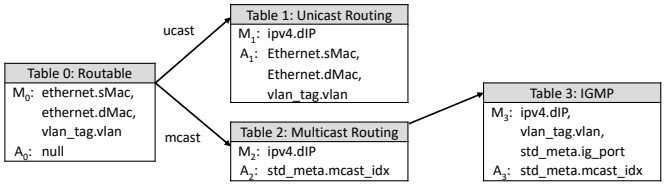
Processors. The dRMT architecture contains N processors. At each clock cycle, each processor can start the following operations for any of its packets: launch *table matches*, launch *actions*, or do nothing (*no-op*). Each table match takes ΔM clock cycles, and each action takes ΔA clock cycles, meaning that an operation dependent on a match or an action has to wait ΔM or ΔA cycles respectively. At each clock cycle, when deciding which operations to launch, the processor is restricted as follows:

(1) It can initiate up to $\bar{M}$ parallel table searches of up to b bits each. For instance, it can look up a match-action table using a key of size $2.5 \cdot b$ by sending three parallel vectors of b bits each, as long as $3 \leq \bar{M}$.
(2) It can modify up to $\bar{A}$ action fields in parallel.
(3) Finally, it can only start matches for up to *IPC* (Inter-Packet Concurrency) different packets, and likewise start actions for up to *IPC* different packets. The set of packets that start matches and the set of packets that start actions need not be equal.

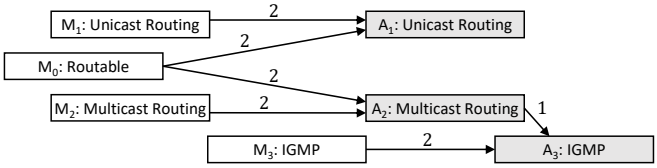
Memory access constraint. At each clock cycle, a P4 table (and hence its associated memory clusters) can only be accessed by a single packet from a single processor.

Crossbar. At each clock cycle, the above constraints mean that each processor can generate up to $\bar{M}$ b-bit-width keys for table lookups that it sends to the memory clusters via a crossbar. The crossbar also permits multicast.

Fixed scheduling. We restrict ourselves to fixed schedules (*i.e.,* predetermined for all types of packets at compile time) given a P4 program and a dRMT architecture. Specifically, we assume that the arriving packets are assigned to one of the N processors in a



(a) Control flow program (fragment from L2L3 program in [22]). Each table i has match M_i and action A_i. Note that table 3 does not match on fields that A_2 modifies, so M_3 does not depend on M_2 or A_2. Yet both A_2 and A_3 modify the same field, and the final value should be that of A_3, so A_3 cannot be executed before A_2.



(b) Operation Dependency Graph (ODG) for the fragmented L2L3 program. Light rectangles represent matches, and dark ones represent actions. Arrows represent dependencies and are annotated with latencies. When scheduling the ODG, we assume that both branches dependent on the unicast vs. multicast condition are executed, similar to RMT's speculative execution model [16].

proc. \ cycle	0	1	2	3	4	5	6	7	8
0	M_0&M_1		M_2&M_3		A_1&A_2	A_3			
1	ΔM	M_0&M_1		M_2&M_3	ΔA	A_1&A_2	A_3		
0			M_0&M_1		M_2&M_3		A_1&A_2	A_3	
1				M_0&M_1		M_2&M_3		A_1&A_2	A_3

(c) Naive schedule with conflicts. At clock cycle 2, two architectural constraints are violated: (1) both the first and the third packet assigned to processor 0 are scheduled to execute matches although IPC = 1, and (2) 4 parallel table searches are initiated although $\bar{M} = 2$.

proc. \ cycle	0	1	2	3	4	5	6	7	8	9
0	M_0&M_1		no-op	M_2&M_3		A_1&A_2	A_3			
1		M_0&M_1		no-op	M_2&M_3		A_1&A_2	A_3		
0			M_0&M_1		no-op	M_2&M_3		A_1&A_2	A_3	
1				M_0&M_1		no-op	M_2&M_3		A_1&A_2	A_3

(d) Schedule without conflicts. The insertion of a no-op (null) in the schedule slightly increases latency but helps resolve both the architectural conflicts.

Figure 4: A simple unicast-multicast packet processing program, its Operation Dependency Graph (ODG), and two potential schedules with and without conflicts. In (c) and (d) each row represents the schedule for a different packet with its allocated processor, while each column represents a different clock cycle. A match lasts $\Delta M = 2$ clock cycles, and an action $\Delta A = 1$ clock cycle. We assume that there is no concurrency between different packets in the same processor ($IPC = 1$) and that at most 2 parallel table searches can be initiated at each clock cycle by a processor ($\bar{M} = 2$).

strict round-robin fashion, and that each processor receives a new packet every P clock cycles. Therefore, the switch throughput is N/P (*e.g.,* $P = N$ means that a new packet enters the switch every cycle). Then, we need to find a *fixed schedule*, *i.e.,* a single cycle-by-cycle schedule that is pre-determined at compile time and is applied in the exact same way to all incoming packets. For instance, at line rate ($P = N$), the same operations that are executed by processor 0 at cycle t are also executed by processor 1 at $t + 1$, by processor 2 at $t + 2$, and so on, until processor 0 executes the same operations again at $t + P$. This schedule is *valid* whenever it satisfies both the P4-specific constraints and the dRMT architectural constraints.

Example 3.2 (Valid schedule). We want to find a fixed schedule with $N = 2$ processors to support a throughput of 1 (*i.e.,* $P = 2$) for the unicast-multicast P4 example described in Example 3.1 and Figure 4b. For simplicity, assume that each match requires $\Delta M = 2$ cycles and each action $\Delta A = 1$ cycle; that the limit $\bar{A}$ on action fields is large and can be ignored; that the limit on parallel table searches is $\bar{M} = 2$, with all the keys of size $\leq b$ bits; and that the packet concurrency is $IPC = 1$, *i.e.,* the matches (resp. actions) that a processor executes in a given cycle are restricted to belong to the same packet.

Figure 4c first illustrates a naive schedule that violates the architectural constraints. The top row tracks time in cycles, and the following rows represent different incoming packets. The leftmost column reflects the processor that services this packet, alternating between processors 0 and 1 in a round-robin manner. To build this first schedule, we simply follow the ODG in Figure 4b from left to right and top to bottom, thus arriving at the following possible sequence of operations:

M_0&$M_1 \rightarrow M_2$&$M_3 \rightarrow A_1$&$A_2 \rightarrow A_3$.

Note that the ODG allows all matches to run in parallel; however, since $\bar{M} = 2$, only two of them can run in parallel. Hence, at time 0, a packet enters processor 0, and M_0 and M_1 are executed in parallel. At time 2, after M_0 and M_1 are finished on this packet, M_2 and M_3 are executed concurrently. All packets in all processors follow the same schedule, hence the entire sequence is simply shifted one column to the right at each row. Unfortunately, this scheduling sequence is invalid because it violates two architectural constraints. First, in clock cycle 2, processor 0 executes matches corresponding to 2 different packets (the first and the third), thus exceeding the IPC limit of 1. Second, in clock cycle 2, processor 0 executes 4 matches while only $\bar{M} = 2$ matches are allowed.

Figure 4d illustrates an alternative schedule without conflicts. The sequence of operations is:

M_0&$M_1 \rightarrow$*no-op* $\rightarrow M_2$&$M_3 \rightarrow A_1$&$A_2 \rightarrow A_3$.

The insertion of the *no-op* in the schedule slightly increases latency but helps resolve all conflicts.

Scheduling objective. Given a dRMT architecture with N processors and a P4 program, *our general objective is to maximize the dRMT throughput*. To do so, at compile time, we run an optimization sub-routine that indicates whether a given throughput is feasible under the constraints provided by the P4 program dependencies and the dRMT architecture. We can then use this subroutine in a binary-search procedure to establish the maximum throughput.

In addition, given any arbitrary dRMT throughput, we want to minimize the system's resources required to support this throughput, and in particular the number of packets that each processor needs to handle. Let T be the schedule's fixed packet latency. Then we find [5] that the maximum concurrent number of packets at each processor is $\lceil T/P \rceil$. Thus, given N and P, if we minimize latency, we also minimize the maximum number of packets that each processor needs to handle. As a result, given some assumed throughput, we formally define our scheduling goal as:

$$\textbf{Minimize} \quad T$$
$$\text{subject to:} \begin{cases} \text{P4 program dependency constraints} \\ \text{dRMT architectural constraints} \\ \text{dRMT throughput} \end{cases} \tag{1}$$

3.2 Simplifying dRMT scheduling

Now that we have established the scheduling goal in Equation 1, we show how it is possible to simplify the dRMT scheduling problem by successively considering only a single processor, then a single packet on a single processor.

Single-processor scheduling. We start by showing that to schedule dRMT, we can focus on scheduling a single processor instead of jointly scheduling all the processors. Specifically, consider a fixed schedule for a single processor that respects the program-specific dependency constraints and the $\bar{M}$, $\bar{A}$ and IPC architectural constraints and is applied by all processors in the exact same manner. Then it is a *valid* dRMT schedule that respects *all* the constraints. In particular, it does not conflict with the other processors when accessing memory clusters. Formally:

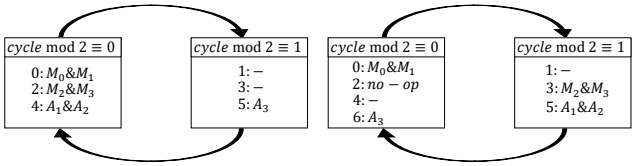
OBSERVATION 1 (CONTENTION-FREE MATCHES). *Consider a single-processor schedule that is applied by all processors. Then there is no memory contention,* i.e., *all match requests initiated by different processors at the same cycle are destined to different memories.*

This observation stems from the fact that all packets arrive at the switch at different clock cycles, and since the schedule is fixed, the time offset from packet arrival to memory access is also fixed. In addition, since the ODG is acyclic, each packet accesses each memory at most once. Therefore, in any given memory cluster, the times at which the cluster's memory is accessed are different for different packets.

dRMT achieves higher throughput than RMT. An important corollary of Observation 1 is that the throughput of the dRMT architecture is guaranteed to be at least the throughput of a corresponding RMT architecture. Intuitively, consider the sequence of operations on a packet in an RMT pipe. Then, we can use the same schedule for each dRMT processor in the same order. Clearly, if a dRMT processor has the same $\bar{M}$ and $\bar{A}$ capacity as an RMT stage, such a schedule will respect the program-specific dependency constraints and the $\bar{M}$, $\bar{A}$ and IPC architectural constraints, and therefore it results in a valid schedule for dRMT. Formally:

THEOREM 3.3. *The throughput of a program on dRMT is at least that of RMT.*

The full proofs of all the results in this paper are available in an online extended version [5].



(a) Naive schedule with conflict that corresponds to Figure 4c. **(b) Schedule without conflicts that corresponds to Figure 4d.**

Figure 5: Illustrating the naive and conflict-free schedules from Figure 4 using a cyclic single-packet analysis. Each rectangle shows the operations that are executed simultaneously by a processor in steady-state, possibly for different packets.

Single-packet scheduling. As mentioned, a fixed schedule applies the same operations in the same order to all incoming packets, which periodically arrive at a given processor every P slots. Consider a packet that arrives at time t. At this time, the processor *simultaneously* executes set of operations 0 for this packet, set P for the packet that arrived at $t - P$, set $2P$ for the packet that arrived at $t - 2P$, and so on. These simultaneously-executed sets are precisely the sets of operations that the first packet itself executes at all cycles that are equivalent to t modulo P, because the first packet will also execute the set P of operations at time $t + P$, the set $2P$ at $t + 2P$, and so on. As a result, instead of analyzing how the different packets share resources at this processor, analyzing all the operations of a single packet will suffice.

Example 3.4 (Single-packet scheduling). Consider again the unicast-multicast example of Figure 4. Figure 5 shows a simplified analysis that considers only a single packet at a single processor. Each rectangle represents an equivalence class, *i.e.,* all the operations that are executed simultaneously in a cycle modulo P. The time slot at which each operation is executed appears before the operation. For a schedule that respects the program dependency constraints to be valid, we only need to make sure that all operations assigned to an equivalence class respect the $\bar{M}$, $\bar{A}$, and IPC constraints. Figure 5a illustrates the schedule with conflicts from Figure 4c. The operations in rectangle *cycle mod* $P \equiv 0$ again violate the same two architectural constraints: there are four match operations where $\bar{M} = 2$ and there are two distinct execution times (0 and 2) for the match operations where IPC = 1 (detailed in observation 3). Figure 5b illustrates the conflict-free schedule with the no-op from Figure 4d. It is easy to verify the validity of the schedule by considering the operations in each rectangle.

As illustrated in Figure 5, in order to establish a *single-packet schedule* that respects the $\bar{M}$, $\bar{A}$ and IPC architectural constraints, we can transform these constraints into cyclic constraints, *i.e.,* constraints modulo P on the scheduling sequence of a single packet. Specifically, we define P equivalence classes that correspond to the schedule period length and rely on the following observation:

OBSERVATION 2 (CYCLIC PROCESSOR SCHEDULE). *Constructing a valid schedule for a single processor corresponds to assigning each match and action operation in the ODG to an equivalence*

class, while ensuring that the requirements of the operations assigned to the same equivalence class do not violate the architectural constraints, i.e., *$\bar{M}$, $\bar{A}$, and IPC.*

It is straightforward to verify that the $\bar{M}$ and $\bar{A}$ constraints are respected by considering all the operations in each of the equivalence classes. However, to verify that the IPC constraint is respected we need one additional observation:

OBSERVATION 3 (PACKET CLASSIFICATION). *The number of different packets that a processor initiates matches (actions) for in cycle t equals the number of distinct execution times of match (action) nodes in equivalence class $t \bmod P$ (*i.e., *the class of all nodes with the same remainder when their execution start times are divided by P).*

Using these observations, we formally obtain the following result:

THEOREM 3.5. *The single-packet schedule is valid iff the full dRMT schedule is valid.*

3.3 Integer linear program

Before presenting our ILP scheduling solution, we establish the following scheduling hardness result:

THEOREM 3.6. *The dRMT scheduling problem is NP-hard.*

Our proof is based on a reduction from the bin-packing problem in which the bins represent cycles modulo P and the objects of different volumes represent action (match) operations.

With the intractability result at hand, we observe that all dRMT architectural constraints are integer valued. Therefore, as long as the target function is linear as well, we can express this scheduling problem as an Integer Linear Programming (ILP) problem, which can be optimally solved using an ILP solver such as Gurobi [6].

ILP formulation. We can restrict ourselves to a cyclic schedule with modulo constraints, as established in Theorem 3.5. Focus on a packet π, and consider an ODG node corresponding to operation v for this packet. We denote by $t(v)$ the time at which operation v starts. Our goal is to find a cyclic schedule for π that satisfies all of the constraints and minimizes the maximum $t(v)$ among all nodes v, *i.e.,* the start time of the last operation. Then, the first set of constraints is $t(v) \leq T \ \forall v \in \text{ODG}$ where T is the objective function we want to minimize.

We next provide an overview on how the ILP deals with the three types of constraints: P4-program dependency constraints (with ΔM and ΔA); resource constraints (with $\bar{M}$ and $\bar{A}$); and inter-packet concurrency constraints (*IPC*).

Dependency constraints. When solving the scheduling problem, we must respect the dependencies specified by the edges in the ODG and their corresponding delay. We can express these constraints as

$$t(v) - t(u) \geq \tau(u,v) \quad \forall (u,v) \in \text{ODG}, \tag{2}$$

where $\tau(u,v)$ is the number of cycles that must pass between u and v (*i.e.,* ΔA or ΔM). Note that we do not demand equality. The price for this flexibility is the scratch pad that may be needed to store intermediate results until they are consumed by a successor node (§5).

Resource constraints. As stated by Observation 2, we seek to partition all the nodes in the ODG among the P equivalence classes while not violating any resource constraints. To that end, we introduce indicator variables that track the usage of resources by each class. We define the equivalence class of a node to be the remainder of this node's execution time divided by the period length P. Accordingly, we introduce $\text{rq}(v,q,r)$, which is a binary variable that respects $\text{rq}(v,q,r) = 1$ iff $t(v) = q \cdot P + r$, and is 0 otherwise. It is easy to see that the indicator variable must satisfy two equalities:

$$\sum_{q,r} \text{rq}(v,q,r) = 1 \quad \forall v \in \text{ODG}, \tag{3}$$

$$t(v) = \sum_{q,r} (q \cdot P + r) \cdot \text{rq}(v,q,r) \quad \forall v \in \text{ODG}. \tag{4}$$

Let V_M be the set of all match nodes v in the ODG, and $k(v)$ be the key size in bits of match node $v \in V_M$. Likewise, let V_A be the set of all action nodes v and $a(v)$ be the number of modified fields by action $v \in V_A$. Now, we can formulate the match and action resource constraints as

$$\sum_{v \in V_M, q} \left\lceil \frac{k(v)}{b} \right\rceil \cdot \text{rq}(v,q,r) \leq \bar{M} \quad \forall r, \tag{5}$$

$$\sum_{v \in V_A, q} \text{a}(v) \cdot \text{rq}(v,q,r) \leq \bar{A} \quad \forall r. \tag{6}$$

IPC constraints. We want to set IPC as an upper limit on the maximum number of different packets for which the processors can generate match keys in the same time slot. To do so, we rely on Observation 3. Specifically, let (q,r) correspond to an execution time $t = q \cdot P + r$. Then, we need to limit the number of distinct values q of match operations that belong to the same class (*i.e.,* same r value). To do so, we introduce an indicator $\text{pm}(q,r)$ for match operations such that if at least one match operation takes place at time $t = q \cdot P + r$ then $\text{pm}(q,r) = 1$. This can be expressed as follows:

$$\sum_{v \in V_M} \text{rq}(v,q,r) \leq \text{pm}(q,r) \cdot \sum_{v \in V_M} 1 \quad \forall q,r. \tag{7}$$

Finally, we can limit the number of different packets for which matches are initiated at the same cycle:

$$\sum_{q} \text{pm}(q,r) \leq IPC \quad \forall r. \tag{8}$$

Namely, we demand that the number of different q values of matches that are executed in the same time slot is bounded by IPC without limiting the number of concurrent matches that belong to the same packet (*i.e.,* same r value). The IPC action constraints are defined in an identical manner.

Accelerating the ILP run-time. We have developed three techniques to accelerate the ILP run-time. These techniques allow us to find a feasible solution to the ILP, which can be used to seed the ILP solver or quickly establish feasibility in the binary-search procedure. In the interest of space, we leave a detailed description of these techniques to the longer version of this paper [5].

4 EVALUATION

We use two metrics to compare RMT with dRMT on packet-processing programs: (1) the minimum number of processors required to sustain line rate (*i.e.,* one packet per clock cycle), and (2) the minimum number of threads required to sustain one packet per clock cycle. A separate thread exists for each packet currently

Parameter	RMT	dRMT
Match capacity ($\bar{M}$)	8	8
Match unit size (b)	80 bits	80 bits
Action capacity ($\bar{A}$)	224	32
Match latency (ΔM)	18	22
Action latency (ΔA)	2	2
Inter-packet concurrency (IPC)	1	1 or 2
Memory disaggregation	Yes	Yes

Table 1: Parameters for RMT and dRMT.

residing at a dRMT processor, for which some state (*e.g.,* the packet header vector) needs to be maintained. For a fixed throughput of one packet per cycle, the number of threads across all processors is exactly the same as the latency of the program.

First, we compare the two architectures on four real P4 programs, three derived from an open-source program, switch.p4 [13], and one proprietary program. Second, because of the paucity of real P4 programs, we compare the two architectures on 100 randomly-generated operation dependency graphs (ODGs). Third, we illustrate how throughput degrades on each architecture as we decrease the number of processors, showing that dRMT does not have a performance cliff, unlike RMT. Fourth, we conclude by reporting on the run-times of the dRMT ILP.

4.1 Experimental setup

We compare four architectures: *RMT*, *fine-grained RMT*, *dRMT with IPC* = 1, and *dRMT with IPC* = 2. Fine-grained RMT allows matches and actions within a single P4 table to be split and placed in different RMT stages. It provides greater flexibility than RMT at the cost of temporarily holding the action result in the packet header—a cost we ignore for RMT. We also compare with a lower bound. The lower bound captures the minimum number of processors needed to support line rate, if the bottleneck resource (either match or action capacity) is fully utilized. It also captures the minimum latency for the program based on its critical path.

Numeric parameters. For both architectures, we assume the number of memory clusters equals the number of processors/stages. We list other parameters in Table 1. We chose these parameters based on the RMT paper and our estimates for match and action latency. dRMT's match latency is higher due to the crossbar. dRMT's action capacity is lower to ensure its chip area is competitive with RMT (we expand on this in §5.1). We do not consider *IPC* = 2 for RMT, as this would require sending two packet headers through the pipeline, effectively doubling the width of the entire datapath across all stages. In dRMT, by contrast, supporting *IPC* = 2 only requires an additional packet buffer and a few muxes in each processor (§6), resulting in only a modest increase in chip area.

Evaluating RMT's performance. For both the fine-grained and default RMT architectures, we formulate an ILP that handles the match and action capacity constraints described above and dependency constraints captured by the operation dependency graphs. Our RMT ILP is similar to that of Jose *et al.* [22], but does not consider per-stage table capacity constraints, in effect simulating an RMT pipeline with fully disaggregated memory. This implies that dRMT's actual improvements relative to RMT—with local, non-shareable memory in each stage—will be higher than the numbers reported here. We use the RMT ILP to calculate the minimum number of pipeline stages S for RMT that satisfies both the capacity and dependency constraints.

Evaluating dRMT's performance. For dRMT, we run the ILP described in §3, using our heuristics [5] to accelerate the ILP's runtime. We use our binary-search procedure described in §3 to calculate the minimum scheduling period P such that a single processor can receive a packet every P clock cycles.

Metrics. For dRMT, if the minimum scheduling period is P, then a single processor can receive a packet at most once every P clock cycles. This implies that at least P processors are required to support a throughput of one packet per clock cycle. For RMT, assuming each stage can process a packet every clock cycle, at least S stages are required to run the program at one packet per clock cycle.

The minimum number of total threads (across all stages) for RMT is obtained by multiplying the minimum number of stages by the sum of the RMT match and action latencies. For dRMT, it is output by the ILP as its optimization objective.

Remark. The ILPs for both RMT and dRMT assume that both branches of a condition are always speculatively executed (see §3). Scheduling mutually exclusive operations together (which cannot simultaneously execute for *any* packet) may reduce the number of processors/stages and threads for both architectures.

4.2 Experimental results

Comparing dRMT with RMT on real P4 programs. Tables 2 and 3 show the minimum number of processors and threads required to support a throughput of one packet per clock cycle on four P4 programs. Three of them are derived from switch.p4, an open-source P4 program [13]. They correspond to switch.p4's ingress pipeline, its egress pipeline, and a combined ingress+egress program that runs on a single shared physical pipeline to improve utilization, as suggested by RMT [16]. The combined switch.p4 program effectively improves utilization through statistical multiplexing; it creates opportunities to run a highly utilized stage from one pipeline with an underutilized stage from the other in the same hardware stage.

We also use a proprietary P4 program from a large switching ASIC manufacturer. This program has 50% more lines of code than switch.p4, implements all but a few of switch.p4's forwarding features, and some that switch.p4 does not have. For anonymity, we normalize the critical path for latency and the lower bound on the number of processors for the proprietary program to one.

The results show that as we progress from RMT towards dRMT (*IPC* = 2), the minimum number of processors and threads both decrease, because more disaggregation enables more flexible scheduling. The results also show how an *IPC* = 2 is important to take advantage of inter-packet parallelism within the same processor. Neither an RMT stage nor a dRMT processor with *IPC* = 1 can exploit this kind of parallelism. *IPC* = 2 also reaches the upper bound on throughput for these programs, showing how a small degree of inter-packet parallelism is sufficient to extract high throughput.

dRMT's greatest gains are on programs that cause an imbalanced pipeline of match and action operations—and hence wasted resources. dRMT compacts such programs (*e.g.,* switch.p4's egress alone) into a smaller number of processors. When the program already has a balanced RMT pipeline (*e.g.,* combining the ingress and egress switch.p4 into one program), dRMT's gains are lower.

P4 program	RMT	RMT fine	dRMT ($IPC = 1$)	dRMT ($IPC = 2$)	Lower bound
ingress	18	17	17	15	15
egress	12	11	11	7	7
combined	22	21	21	21	21
proprietary	2.0	2.0	2.0	1.0	1.0

Table 2: Minimum number of processors to achieve line rate on each architecture. The lower bound for the proprietary program is normalized to one for anonymity.

P4 program	RMT	RMT fine	dRMT ($IPC = 1$)	dRMT ($IPC = 2$)	Critical path
ingress	360	340	245	243	243
egress	240	220	217	198	197
combined	440	420	243	243	243
proprietary	2.82	2.82	1.04	1.01	1.0

Table 3: Minimum number of threads to achieve line rate on each architecture. The critical path latency numbers are based on ΔM and ΔA for dRMT and provide a lower bound on the number of threads necessary. The critical path for the proprietary program is normalized to one for anonymity.

Comparison on random ODGs. To compare dRMT with RMT on a larger variety of possible P4 programs, we generate random ODGs based on the characteristics of switch.p4's ODG. Specifically, we generate 100 different ODGs that reflect different P4 programs of varying size. For each ODG, we report the minimum number of processors/stages required to support the corresponding program at line rate. We generate the ODGs as follows:

(1) Generate a random directed acyclic graph with 100 nodes. An edge (i, j) where $i < j$ exists with probability p. p is chosen so that the total number of edges is 500 on average.
(2) Then, the nodes are visited according to a topological sort. Each non-leaf node is randomly chosen to be either a default action with probability 0.15; or a conditional node, *i.e.,* a single-field action node representing a predicate, with probability 0.25; or split into a match node followed by an action node with probability 0.6. Likewise, each leaf node is either a default action with probability 0.15, or split into a match node followed by an action node with probability 0.85.
(3) For a non-conditional action node, the number of fields is first sampled from a geometric distribution with a mean of 4 fields and then truncated to the interval $[1, 32]$.
(4) For a match node, the key width is first sampled from a geometric distribution with a mean of 106 bits and then truncated to the interval $[80, 640]$.

All the parameters above, *i.e.,* number of edges, probabilities for node types, and parameters for the truncated geometric distribution are chosen based on distributions observed in switch.p4. We choose a geometric distribution to capture our empirical observation from switch.p4 that higher key widths or action fields are less likely.

Figure 6 illustrates the results. As expected, dRMT with $IPC = 2$ provides more flexibility and results in the lowest number of required processors. dRMT with $IPC = 1$ follows and shows a consistent advantage over RMT. Across all 100 randomly generated graphs, dRMT with $IPC = 2$ had an average reduction of 10% in the number of processors relative to RMT and a maximum of 30%.

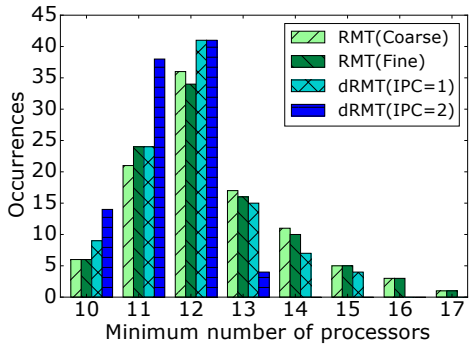


Figure 6: Histogram showing the minimum number of processors for dRMT and RMT on 100 random ODGs.

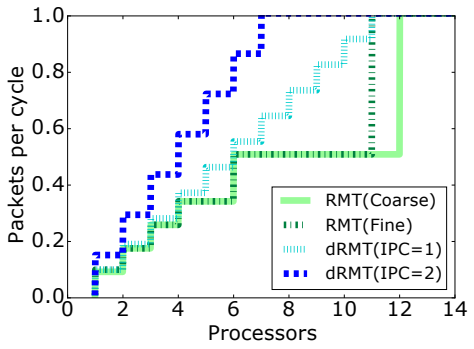


Figure 7: Throughput on switch.p4's egress pipeline. dRMT's performance scales linearly with processors and achieves the lower bound, while RMT displays performance cliffs.

dRMT eliminates performance cliffs. Once we determine both S and P, which respectively enable RMT and dRMT to enable a throughput of one packet per clock cycle, we can calculate the throughput $th(N)$ as a function of the number of processors N. [2]

$$th_{RMT}(N) = \min(1, 1/(\lceil S/N \rceil)) \quad (9)$$

$$th_{dRMT}(N) = \min(1, N/P) \quad (10)$$

The throughputs are capped at one because it is challenging to build on-chip memories to support two or more reads/writes per clock cycle—and hence two or more packets in a cycle.

Figure 7 illustrates the effect of decreasing the number of processors on the throughput. It plots $th(N)$ for each architecture using switch.p4's egress pipeline. The figure illustrates the performance cliff for both RMT variants and the linear degradation in throughput for dRMT. The results for switch.p4 ingress and combined are not shown, but are similar. For example, for the ingress program, RMT's throughput drops to 50% as the number of processors decreases to 17 from 18.

dRMT ILP run-time evaluation. We measure the time taken by the dRMT ILP as a function of the number of processors N, while targeting a fixed throughput of one packet per clock cycle, using the

[2]We assume the ability to recirculate packets back into the pipeline in RMT while reducing throughput by a factor of the number of recirculations.

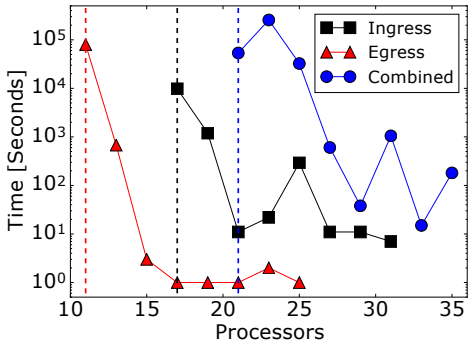


Figure 8: ILP run-time in dRMT ($IPC = 1$) as a function of the number of processors for a throughput of one packet per clock cycle. The dashed lines show the minimum number of processors required to run at one packet per clock cycle. The non-monotonicity is an artifact of the ILP's discrete-valued nature.

three open-source switch.p4 programs. We carried out our measurements on an HP ProLiant DL785G5 machine with 8 AMD quad-core processors (2.2 GHz), each with a shared 2 MB L3 cache, and 256 GB DDR2 RAM at 533 MHz.

Figure 8 depicts the results for dRMT with $IPC = 1$. The ILP run-time drops quickly (by two to three orders of magnitude) as soon as the number of processors is slightly larger than the minimum number necessary to run the program at one packet per clock cycle. The reason is that the ILP is much easier to solve when there is a little bit of slack. We do not show dRMT with $IPC = 2$ because the run-times with $IPC = 2$ never exceeded a few minutes regardless of the number of processors. This is because $IPC = 2$ makes the scheduling problem easier by providing more flexibility.

5 HARDWARE ARCHITECTURE

The dRMT architecture consists of a set of match-action processors, connected to a set of memory clusters through a crossbar. Each match-action processor contains processing elements to (1) generate match keys for matches and (2) update packet fields for actions. While the overall architecture of dRMT is different from RMT, the processing elements for matches and actions are similar. Hence, we adopt the design of these processing elements as is from RMT.

The key difference between RMT and dRMT is that dRMT runs a packet to completion once it is assigned to a processor, instead of moving it from stage to stage. While run-to-completion provides dRMT with more flexibility (§4), it also requires each processor to store the entire program. This is in contrast to RMT, where each pipeline stage only stores the fragment of the program that is executed within that pipeline stage. As a result, dRMT's hardware design requires some optimization to avoid additional cost in digital logic—and therefore silicon area and power. This section focuses on the optimizations within the individual match-action processors (§5.1), the memory clusters (§5.2), and the shared crossbar (§5.3) connecting processors to clusters.

We mention many constants in this section, such as the number of bits in the packet vector, the number of search key bits that can be issued by a processor every clock cycle, *etc.* Unless otherwise

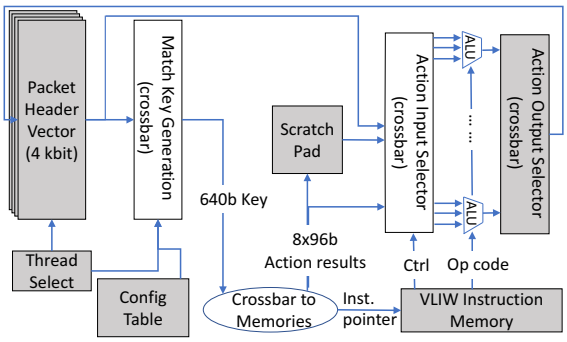


Figure 9: The design of a dRMT processor.

stated, we choose these design parameters to be the same as the published RMT architecture [16]. This enables a more straightforward comparison between our work and theirs.

5.1 Match-action processors

Each match-action processor admits one packet every P clock cycles, where P is the scheduling period. If the maximum program latency is T clock cycles, each processor must receive and process up to $m = \lceil T/P \rceil$ packets in parallel. Hence, the processor uses an m-threaded design to store packets awaiting service, each thread corresponding to a different packet. This is similar to hardware multithreading in general purpose CPU cores and GPUs [24], with the same purpose: to achieve high utilization of the processor in the face of high latency operations (*e.g.,* L1 cache misses for general purpose CPUs, table matches for dRMT).

Figure 9 shows the processor's architecture. We list its components below.

(1) Packet header vectors to store packet headers and metadata, *i.e.,* data about the packet that was derived from the packet and/or table lookup contents, but is not part of the packet itself.
(2) Match table key generation logic to generate match keys for table lookups.
(3) Action Arithmetic Logic Units (ALUs) that allow multiple packet fields to be modified in parallel every clock cycle.
(4) A Very Large Instruction Word (VLIW) [16] instruction memory that specifies the configuration (opcode and operands) for each action ALU on every clock cycle.
(5) An action input crossbar to select inputs for each ALU.
(6) An action output crossbar to route ALU outputs to the packet header vectors after modification.
(7) A scratch pad to temporarily store associated data returned as a result of a table match (which we term action data segment), if it is only going to be used in a later clock cycle.
(8) A thread scheduling configuration table to help select the thread for match and action operations in every clock cycle. Selecting multiple threads provides opportunities for inter-packet concurrency, as discussed earlier.

Match key generation. Based on RMT, we use a 4-kbit packet header vector, structured as 64 8-bit fields, 96 16-bit fields, and 64 32-bit fields, for a total of 224 fields. From this vector, we extract

fields to send up to 640 bits of match search key to the crossbar. This is structured as eight 80-bit key segments, *e.g.,* it can consist of one 320-bit key spanning 4 segments, and two 160-bit keys spanning 2 segments each. We do not find the need to form separate 640-bit keys for the hash tables and TCAMs and find that we can generally fit both the hash table and TCAM keys generated from a processor into 640 bits.

In RMT, the match keys are generated by a 4-kbit-to-640-bit crossbar that takes in the packet header vector as input and outputs up to eight search keys. Each match-action stage needs only one configuration setting for the crossbar, stored in that stage in flip-flops. Different stages have independent configurations so each can generate different search keys depending on what is being looked up in each stage.

On the contrary, dRMT is a run-to-completion architecture, so each processor must be able to generate different search keys in each of the P clock cycles of its repeating schedule. To reduce chip area, instead of storing an identical set of P search-key crossbar configuration settings in each processor, we store one copy of the complete configuration in a P-entry table in a central chip location, and then distribute the configuration to each processor just in time for key generation, via a two-level distribution tree.

Recall that dRMT can schedule match/action operations from multiple threads in the same clock cycle (IPC $>$ 1; see §3). The only restriction is that the combined search keys from all the threads fit into the 640-bit search key.

ALUs. The goal of each arithmetic and logic unit (ALU) is to perform one operation per clock cycle on one or two packet header fields, such as arithmetic on two 32-bit numbers, left or right shifts, comparisons, or logic operations. Both in RMT and dRMT, ALUs themselves are relatively cheap to implement, but can require expensive logic around them to get operands and outputs into and out of the ALUs. For instance, both in RMT and dRMT, the crossbar logic required to extract ALU inputs/operands from the 4-kbit packet header vector and the 768-bit action data segments (*i.e.,* the match results from up to eight tables, each of which is 96 bits) is considerable.

In addition, while in RMT, each stage only needs instruction memory for the program fragment that is resident on that stage, in dRMT, each processor needs to store the entire program. This requires us to reduce the number of ALUs in the dRMT design, because a larger number of ALUs increases the width of the VLIW instruction. RMT uses 224 ALUs within its VLIW instruction, one for each of the 224 fields in the 4-kbit packet header vector. Based on our analysis of switch.p4 [13], we find that 32 parallel ALUs are sufficient. Figure 10 illustrates this analysis of switch.p4. We note that over 90% of the tables require eight or fewer primitive actions [12] to be performed for their largest compound action, and over 90% of the primitive actions used can be performed with a single ALU. Analysis of a proprietary P4 program gives similar results.

The RMT designers justify their choice of an ALU for every field by noting: "Allowing each logical stage to rewrite every field may seem like overkill, but it is useful when shifting headers... A logical MPLS stage may pop an MPLS header, shifting subsequent MPLS headers forward" [16]. For the particular use case mentioned,

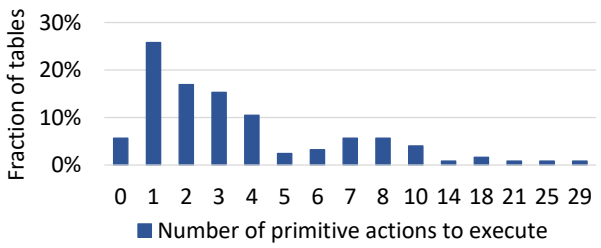


Figure 10: Primitive actions required for switch.p4 tables.

an array of 6 MPLS headers in P4 requires only 6 32-bit ALUs to implement a push or pop operation. Larger header arrays are uncommon in networking protocols. If such cases are encountered in a P4 program that cannot be handled with a single VLIW instruction for 32 ALUs, dRMT with 32 ALUs can perform it in multiple cycles.

dRMT's ALUs use 32-bit inputs and outputs and are functionally identical to the ALUs on 32-bit fields in the RMT chip. With 32 such ALUs, we can modify up to 1 kbit in the packet vector in one clock cycle. Because we only have 32 ALUs, we implement a 32×224 crossbar to write back the ALU outputs to 32 out of 224 locations in the packet header vector. This output crossbar does not exist in RMT because each of the 224 fields has one ALU hard-wired to it.

Recall that up to eight matches may execute every clock cycle, so we can have up to eight distinct actions executing every clock cycle. In aggregate, these eight actions can use up to 32 ALUs. The compiler ensures that the eight actions are implemented using disjoint sets of ALUs in order to avoid resource conflict. Thus, each of the 32 ALUs is assigned to one of the eight actions every clock cycle; this action configures that ALU with an operand and an opcode during that cycle.

VLIW instruction memory. The VLIW instruction memory is used to configure each of the 32 ALUs within a dRMT processor by supplying it with opcodes and operands every clock cycle. We implement the instruction memory as 32 per-ALU SRAM *slices* (Figure 11). Each slice can hold up to 1K entries; each entry corresponds to the configuration of that ALU for one of 1K different actions. We chose the number 1K to provide the same number of actions allowed by RMT (32 stages * 32 user-defined actions per stage). Each entry within the slice stores the opcode for the ALU along with the address of the operand. The operand can be one of the 224 packet fields, one of the eight 96-bit action data segments (*e.g.,* the value of the next hop for an action that sets the next hop) returned as a result of the table lookup, a constant, or data from the scratchpad.

Which of the 1K entries within a slice configures an ALU on every cycle? Each table lookup, in addition to returning a data segment, also returns a 10-bit instruction pointer that represents one of the 1K different actions. These instruction pointers returned from a table lookup address the per-ALU slices during that cycle. To determine which of the 32 per-ALU slices a particular instruction pointer should address on a particular clock cycle, we use a 32×96b configuration table. This table is indexed by a 5-bit program counter and hence has 32 entries. Each entry in the table consists of 32 3-bit select fields, one for each ALU. The entry indicates which of the eight actions (and hence which of the 8 10-bit instruction pointers) executing in that cycle are assigned to each of the 32 ALUs for that cycle.

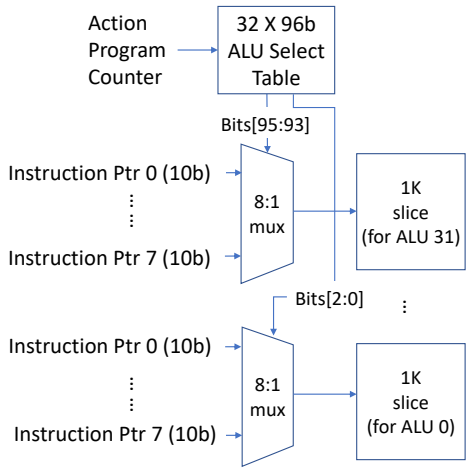


Figure 11: VLIW instruction memory.

The VLIW instruction memory cannot be replaced by a centralized distribution tree like the match configuration. This is because the VLIW ALUs' opcodes need to be determined at run-time and potentially could be different for every packet, depending on the instruction pointer returned by the match-action table lookup for that packet. On the other hand, the match crossbar configuration can be determined at compile time and simply needs to be sent to each processor just before the processor needs it.

Scratch pad. If the program only consists of a fixed pattern of consecutive match-immediately-followed-by-action elements, the action data segments are consumed as they are returned. However, in dRMT there are additional challenges:

- The program scheduler may decide to delay an action using no-ops to assign the clock cycle to another packet;
- The scheduler may decide to interleave matches and actions, *e.g.,* $M_1 M_2 \cdots A_1 A_2$, where M_i corresponds to matches and A_i to actions;
- The scheduler may even merge actions following multiple match operations to reduce latency, *e.g.,* $M_1 M_2 \cdots A_{1+2}$, where A_{1+2} corresponds to a merged action that runs both actions A_1 and A_2 in parallel.

While this flexibility improves the scheduling outcome, it requires temporary storage for the results of match operations that are returned from the memory clusters back to the processors, since these results are not always immediately consumed.[3] We implement a *scratch pad* for this purpose. The width of the scratch pad is 96 bits (the width of the action data segment) and it can store up to 64 entries, which we found to be adequate for programs we evaluated. The scratch pad has eight write ports and eight read ports. This allows the processor to write data from eight action results in parallel and later read up to eight action data segments in parallel.

A deeper scratch pad leaves more flexibility for the scheduler to postpone actions, but adds to the hardware cost. If the compiler is given a program where this number of scratch-pad entries is insufficient, the scheduler can rearrange the sequence of matches and actions to reduce the temporary storage required. This can be achieved by, for example, restricting the allowed number of interleaved match-then-action operations (*e.g.,* by changing Equation (2) to an equality instead of an inequality for some dependency constraints, thus leaving less slack), possibly at the expense of a higher program latency, and even lower throughput if a feasible schedule with these new restrictions cannot be found anymore. We leave this refinement to our ILP formulation (§3.3) to future work.

[3]The results of actions are always written back into the packet header vector, and therefore do not need temporary storage.

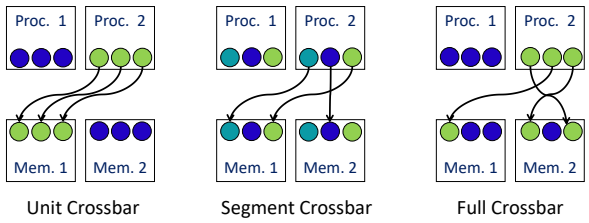


Figure 12: Unit, segment, and full crossbars. This figure represents one specific configuration of each of the three crossbars, not the entire wiring diagram.

5.2 Memory cluster

A dRMT memory cluster is organized like the memory within a single RMT stage. Each memory cluster has a set of memory blocks that can be grouped together to form wider or deeper logical tables. The parameters for memory blocks are identical to RMT [16]. The number of processors is typically equal to the number of memory clusters, but the architecture allows these numbers to differ (§2.1).

5.3 Crossbar

The crossbar connects processors to memory clusters. Every clock cycle, a processor can produce up to $\bar{M} = 8$ key segments of $b = 80$ bits, and expect eight 10-bit instruction pointers plus eight 96-bit action data segments from the memory clusters. The crossbar configuration is statically programmed during compilation, based on the result of the scheduling algorithm.

Crossbar types. When designing the dRMT architecture, we considered various crossbar types and their tradeoffs (Figure 12).

(1) *Unit Crossbar:* One-to-one connectivity between a processor and a memory cluster.
(2) *Segment Crossbar:* One-to-one connectivity between a key (or action data) segment at the k^{th} index on a processor and a segment at the k^{th} index on a memory cluster. This is equivalent to k parallel unit crossbars, one for each segment.
(3) *Full Crossbar:* One-to-one connectivity between any segment on a processor and any segment on a memory cluster. This gives us complete flexibility in terms of table allocation on memory clusters.

In addition, for each of the above-mentioned types, we considered one-to-many multicast crossbars. A one-to-many crossbar enables the processor to access multiple memory clusters at once without incurring extra latency. This allows large tables to be spread across multiple clusters, and still be accessed in a single match operation.

While the segment crossbar seems to be more constrained than the full crossbar, we can prove the following powerful result:

	Unit Crossbar	Segment Crossbar	Full Crossbar
One-to-One	0.544	0.561	4.448
One-to-Many	0.561	0.576	4.464

Table 4: 32x32 crossbar synthesis gate-only area (mm^2).

Component	RMT	dRMT (*IPC* = 1)	dRMT (*IPC* = 2)
Key Generation			
Match key config. register	0.007	0.004	0.005
Match key xbar	0.098	0.049	0.071
Packet Storage			
Packet header vectors	0.110	0.326	0.470
Scratch pad	N/A	0.051	0.051
Actions			
Action input selector	0.486	0.171	0.315
ALUs	0.170	0.071	0.071
Action output selector	N/A	0.048	0.048
VLIW instruction table	0.372	0.336	0.336
Total	1.243	1.056	1.367

Table 5: Area per processor (mm^2).

THEOREM 5.1. *The segment crossbar is equivalent to a full crossbar if: (1) no match tables are split across memory clusters and (2) each processor sends, and respectively each memory cluster receives, at most $\bar{M}$ key segments.*

While we can construct examples to show that the two crossbars are not equivalent when we have a table split across memory clusters, the theorem indicates that for most practical cases, there is no difference between them. We picked the multicast segment crossbar because it comes quite close to the full crossbar in expressiveness, while consuming much lesser area and power. Table 4 details the synthesis numbers for gate area without wiring (Table 6 has the area for a segment crossbar including wires; including wiring, the area ratios between the different crossbars should stay about the same.).

6 HARDWARE COST

We now evaluate the cost of dRMT's hardware implementation and compare it to the RMT design. Currently, the most mainstream process technology for this type of high-performance chip is 16 nm. As mentioned before, we do not have an implementation of dRMT. In order to obtain silicon area estimates, we coded sample logic and synthesized it with the Synopsys Design Compiler for a 1.2 GHz clock cycle target. We demonstrate that the chip area and power differences between the dRMT and RMT chip are small. Our calculations are for the processor portions of RMT and dRMT only, and we do not discuss the chip design elements that are common to both, such as memory clusters, SerDes, Ethernet MAC logic and packet buffer memory. Discussion of the relationship of chip area and cost can be found in chapter 1 of [24]. Table 5 summarizes the area of the major components within the processor.

We synthesized a design for dRMT ALUs with an area close to the one reported in the table. Since we do not know the complete set of ALU operations used by the RMT architecture, our reported area for RMT is based on RMT's estimate [16] that ALUs take up 7% of the entire chip for a 32-stage pipeline with 224 ALUs per stage.

Number of processors	RMT	Crossbar with 32 memory clusters	dRMT (*IPC* = 1) plus crossbar	dRMT (*IPC* = 2) plus crossbar
16	19.9	0.857	17.7	22.7
24	29.9	1.254	26.6	34.1
32	39.8	1.740	35.5	45.5

Table 6: Area for all processors plus interconnect (mm^2).

Number of processors	Power for crossbar with 32 memory clusters
16	0.88 W
24	1.31 W
32	1.75 W

Table 7: Crossbar power.

We estimate an area of 200 mm^2 for RMT's entire chip based on the lower limit from Gibb *et al.* [19], and then scale the area down from a 28 nm to a 16 nm process (We must make some assumptions here in order to compare our areas to theirs, because RMT does not report absolute area numbers for its ALUs.). We estimate 42% of the area of RMT's 224 ALUs. This is more than the fraction of total ALUs involved (*i.e.,* $\frac{32}{224}$) to account for the larger size of dRMT's 32-bit ALUs relative to RMT's 8-bit and 16-bit ALUs.

The packet header vectors in RMT are a simple shift register of 20 packet vectors per stage, to cover 18 cycles of match latency plus 2 cycles of action latency. Instead, in dRMT, the match latency is 22 cycles to cover the additional 4 clock cycles of match latency needed to traverse the crossbar. In addition, in our scheduling experiments with P4 programs, we saw several instances that required up to 29 packets (*i.e.,* threads) per processor to accommodate the schedule because of no-ops. This is higher than RMT because we achieved a higher utilization of the processor resources. Therefore, we assume up to 32 threads per processor in our dRMT architecture, corresponding to 32 packet vectors per processor.

The dRMT packet header vectors cost more area per bit of storage primarily due to the additional read and write ports required. For *IPC* = 2, we need to read 2 different vectors per clock cycle to construct match keys, plus 2 more to perform actions on them, and write back modified vectors for another 2 packets. This totals 4 read ports and 2 write ports.

Table 6 presents the total area for multiple processors. It also gives the area for a multicast segment crossbar (§5.3) that interconnects the processors and memory clusters, sized for the given number of processors and 32 memory clusters. We found that about a third of the area is used by gates (Table 4) and two-thirds by wiring.

We have existing commercial designs containing similar crossbars of up to 16 processors and 16 memory clusters with similar utilization of 33%. We have carefully analyzed techniques such as manually routing crossbar wiring over SRAMs in the memory clusters that should allow us to scale to larger 32×32 crossbars (details are in our extended version [5]).

Table 7 estimates the crossbar's power. This was obtained using Synopsys PrimeTime in the same 16 nm technology, 1.2 GHz clock frequency, 0.9 volts, and 50% switching factor, *i.e.,* the worst case of all data bits changing values every clock cycle. To put our area

and power numbers in context, commercial switch ASICs occupy between 300 and 700 mm^2, and consume between 150 and 350 W.

7 RELATED WORK

dRMT's design is similar to network processing units (NPU) [4, 8–11, 23, 25] and multi-core software routers [17], which feature an array of processor cores with shared memory. Like both CPU and NPU cores, each dRMT core has local instruction memory and a scratchpad for data. However, NPUs lack deterministic guarantees on performance and have historically been slower than line-rate switches. An NPU's non-determinism arises from multiple factors: cache misses, contention within the processor-memory interconnect, pipeline flushes in each core, etc. dRMT's custom-designed crossbar is scheduled at compile time to eliminate all contention—and hence non-determinism. Further, by basing dRMT's VLIW instruction set on RMT, dRMT exploits the parallelism available within packet processing more effectively than NPUs, which suffer a performance hit because their instruction sets resemble conventional CPUs.

Cavium's XPliant [3, 14] and Barefoot's Tofino [2] are two commercial products that support programmability at multi-Tbit/s speeds. Based on publicly available documents [2, 14], it appears both use a pipelined approach similar to RMT. It is unclear from these sources whether they use a crossbar between the pipeline and table memories. If so, the crossbar would introduce a similar additional area and power cost as we have analyzed for dRMT, while providing memory disaggregation alone, *e.g.,* better utilization of the pipeline's processing resources in the presence of large tables. However, both the Xpliant and Tofino architectures would still suffer a performance cliff if a single pass through the pipeline was insufficient.

Prior work [22] has looked at compiling P4 programs to the RMT architecture using an ILP formulation. This ILP formulation needs to handle the memory allocation problem for logical P4 tables while respecting dependencies between these tables. dRMT decouples the memory allocation problem from compute scheduling using the crossbar, essentially reducing the compilation problem to two separate ILPs for memory allocation and compute scheduling.

The problem of *cyclic scheduling* has been studied in the operations research community [18, 20, 21]. In a cyclic scheduling problem, a set of tasks needs to be executed an infinite number of times, while still respecting task dependencies and resource constraints. The objective in cyclic scheduling is to maximize the steady-state throughput, *i.e.,* how frequently an instance of the same task can be executed. Our problem setting is similar, the tasks corresponding to match or action operations. Our formulation differs from the standard cyclic scheduling problem by incorporating a constraint unique to packet processing: we limit the number of packets that can be processed concurrently using the *IPC* parameter.

8 CONCLUSION

This paper presented dRMT, a new architecture for high-speed programmable switching. At the core of dRMT is disaggregation in two forms: in memory disaggregation, we move memories out of processors and into a shared memory pool, while in compute disaggregation, we allow each processor to execute matches and actions in any order respecting program dependencies. Our discussion of disaggregation has been grounded in the context of dRMT, but it is more broadly applicable. For instance, retaining the RMT pipeline but adding a shared memory pool improves RMT's memory utilization. Similarly, disaggregating the matches and actions within a single RMT table and putting them in different stages (as in the RMT-fine architecture) reduces RMT's stage count.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Nate Foster, and our anonymous reviewers for their helpful suggestions. This work was partly supported by NSF grants CNS-1563826, CNS-1526791, and CNS-1617702, a gift from the Cisco Research Center, the Hasso Plattner Institute Research School, the Israel Ministry of Science and Technology, the Gordon Fund for Systems Engineering, the Technion Fund for Security Research, the Israeli Consortium for Network Programming (Neptune), and the Shillman Fund for Global Security.

REFERENCES

[1] A Deeper Dive Into Barefoot Networks Technology. http://techfieldday.com/appearance/barefoot-networks-presents-at-networking-field-day-14.
[2] Barefoot: The World's Fastest and Most Programmable Networks. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf.
[3] Cavium Attacks Broadcom in Switches. http://www.eetimes.com/document.asp?doc_id=1323931.
[4] Cisco QuantumFlow Processor. https://newsroom.cisco.com/feature-content?type=webcontent&articleId=4237516.
[5] dRMT project. http://drmt.technion.ac.il.
[6] Gurobi Optimization. http://www.gurobi.com.
[7] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.
[8] Intel IXP2800 Network Processor. http://www.ic72.com/pdf_file/i/587106.pdf.
[9] IXP4XX Product Line of Network Processors. http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html.
[10] Mellanox Indigo NPS-400 400Gbps NPU. http://www.mellanox.com/page/products_dyn?product_family=241&mtag=nps_400.
[11] Netronome Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx.
[12] P4 Specification. https://p4lang.github.io/p4-spec.
[13] switch.p4. https://github.com/p4lang/switch/tree/master/p4src.
[14] XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.
[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, July 2014.
[16] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.
[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. *ACM SOSP*, 2009.
[18] D. L. Draper, A. K. Jonsson, D. P. Clements, and D. E. Joslin. Cyclic scheduling. In *IJCAI*, 1999.
[19] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ANCS*, 2013.
[20] C. Hanen. Study of a NP-hard cyclic scheduling problem: The recurrent job-shop. *European journal of operational research*, 1994.
[21] C. Hanen and A. Munier. A study of the cyclic scheduling problem on parallel processors. *Discrete Applied Mathematics*, 1995.
[22] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*, 2015.
[23] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Transactions on Networking*, 20(6):1895–1909, 2012.
[24] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, 4th Edition: The Hardware/Software Interface*. 2008.
[25] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. In *ISCA*, 2003.