Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN (Extended Version)

Pat Bosshart[†], Glen Gibb[‡], Hun-Seok Kim[†], George Varghese[§], Nick McKeown[‡], Martin Izzard[†], Fernando Mujica[†], Mark Horowitz[‡]
[†]Texas Instruments [‡]Stanford University [§]Microsoft Research
pat.bosshart@gmail.com {grg, nickm, horowitz}@stanford.edu
varghese@microsoft.com {hkim, izzard, fmujica}@ti.com

ABSTRACT

In Software Defined Networking (SDN) the control plane is physically separate from the forwarding plane. Control software programs the forwarding plane (e.g., switches and routers) using an open interface, such as OpenFlow. This paper aims to overcomes two limitations in current switching chips and the OpenFlow protocol: i) current hardware switches are quite rigid, allowing "Match-Action" processing on only a fixed set of fields, and ii) the OpenFlow specification only defines a limited repertoire of packet processing actions. We propose the RMT (reconfigurable match tables) model, a new RISC-inspired pipelined architecture for switching chips, and we identify the essential minimal set of action primitives to specify how headers are processed in hardware. RMT allows the forwarding plane to be changed in the field without modifying hardware. As in OpenFlow, the programmer can specify multiple match tables of arbitrary width and depth, subject only to an overall resource limit, with each table configurable for matching on arbitrary fields. However, RMT allows the programmer to modify *all* header fields much more comprehensively than in OpenFlow. Our paper describes the design of a 64 port by 10 Gb/s switch chip implementing the RMT model. Our concrete design demonstrates, contrary to concerns within the community, that flexible OpenFlow hardware switch implementations are feasible at almost no additional cost or power.

1. INTRODUCTION

To improve is to change; to be perfect is to change often. — Churchill

Good abstractions—such as virtual memory and time-sharing—are paramount in computer systems because they allow systems to deal with change and allow simplicity of programming at the next higher layer. Networking has progressed because of key abstractions: TCP provides the abstraction of connected queues between endpoints, and IP provides a simple datagram abstraction from an endpoint to the network edge. However, routing and forwarding *within* the network remain a confusing conglomerate of routing protocols (e.g., BGP, ICMP, MPLS) and forwarding behaviors (e.g., routers, bridges, firewalls), and the control and forwarding planes remain intertwined inside closed, vertically integrated boxes.

Software-defined networking (SDN) took a key step in abstracting network functions by separating the roles of the control and forwarding planes via an *open* interface between them (e.g., OpenFlow [27]). The control plane is lifted up and out of the switch, placing it in external software. This programmatic control of the forwarding plane allows network owners to add new functionality to their network, while replicating the behavior of existing protocols. OpenFlow has become quite well-known as an interface between the control plane and the forwarding plane based on the approach known as "Match-Action". Roughly, a subset of packet bytes are matched against a table; the matched entry specifies a corresponding action(s) that are applied to the packet.

One can imagine implementing Match-Action in software on a general purpose CPU. But for the speeds we are interested in—about 1 Tb/s today—we need the parallelism of dedicated hardware. Switching chips have remained two orders of magnitude faster at switching than CPUs for a decade, and an order of magnitude faster than network processors, and the trend is unlikely to change. We therefore need to think through how to implement Match-Action in hardware to exploit pipelining and parallelism, while living within the constraints of on-chip table memories.

There is a natural tradeoff between programmability and speed. Today, supporting new features frequently requires replacing the hardware. If Match-Action hardware permitted (just) enough reconfiguration in the field so that new types of packet processing could be supported at run-time, then it would change how we think of programming the network. The real question here is whether it can be done at reasonable cost without sacrificing speed.

Single Match Table: The simplest approach is to abstract matching semantics in what we call the SMT (Single Match Table) model. In SMT, the controller tells the switch to match any set of packet header fields against entries in a single match table. SMT assumes that a parser locates and extracts the correct header fields to match against the table. For example, an Ethernet packet may have an optional MPLS tag, which means the IP header can be in two different locations. The match is a binary exact match when all fields are completely specified, and is ternary for matches where some bits are switched off (wildcard entries). Superficially, the SMT abstraction is good for both programmers (what could be simpler than a single match?) and implementers (SMT can be implemented using a wide Ternary Content Addressable Memory (TCAM)). Note that the forwarding data plane abstraction has the most rigorous hardware implementation constraints because forwarding is often required to operate at about 1 Tb/s.

A closer look, however, shows that the SMT model is costly in use because of a classic problem. The table needs to store every *combination* of headers; this is wasteful if the header behaviors are orthogonal (the entries will have many wildcard bits). It can be even more wasteful if one header match affects another, for example if a match on the first header determines a disjoint set of values to match on the second header (e.g., in a virtual router [10]), requiring the table to hold the Cartesian-product of both.

Multiple Match Tables: MMT (Multiple Match Tables) is a natural refinement of the SMT model. MMT goes beyond SMT in an important way: it allows multiple smaller match tables to be matched by a subset of packet fields. The match tables are arranged into a pipeline of stages; processing at stage j can be made to depend on processing from stage $i < j$ by stage i modifying the packet headers or other information passed to stage j. MMT is easy to implement using a set of narrower tables in each stage; in fact, it is close enough to the way existing switch chips are implemented to make it easy to map onto existing pipelines [2,14,23,28]. Google reports converting their entire private WAN to this approach using merchant switch chips [13].

The OpenFlow specification transitioned to the MMT model [31] but does not mandate the width, depth, or even the number of tables, leaving implementors free to choose their multiple tables as they wish. While a number of fields have been standardized (e.g., IP and Ethernet fields), OpenFlow allows the introduction of new match fields through a user-defined field facility.

Existing switch chips implement a small (4–8) number of tables whose widths, depths, and execution order are set when the chip is fabricated. But this severely limits flexibility. A chip used for a core router may require a very large 32-bit IP longest matching table and a small 128 bit ACL match table; a chip used for an L2 bridge may wish to have a 48-bit destination MAC address match table and a second 48-bit source MAC address learning table; an enterprise router may wish to have a smaller 32-bit IP prefix table and a much larger ACL table as well as some MAC address match tables. Fabricating separate chips for each use case is inefficient, and so merchant switch chips tend to be designed to support the superset of all common configurations, with a set of fixed size tables arranged in a pre-determined pipeline order. This creates a problem for network owners who want to tune the table sizes to optimize for their network, or implement *new* forwarding behaviors beyond those defined by existing standards. In practice, MMT often translates to *fixed* multiple match tables.

A second subtler problem is that switch chips offer only a limited repertoire of actions corresponding to common processing behaviors, e.g., forwarding, dropping, decrementing TTLs, pushing VLAN or MPLS headers, and GRE encapsulation. And to date, OpenFlow specifies only a subset of these. This action set is not easily extensible, and is also not very abstract. A more abstract set of actions would allow any field to be modified, any state machine associated with the packet to be updated, and the packet to be forwarded to an arbitrary set of output ports.

Reconfigurable Match Tables: Thus in this paper, we explore a refinement of the MMT model that we call RMT (Reconfigurable Match Tables). Like MMT, ideal RMT would allow a set of pipeline stages each with a match table of arbitrary depth and width. RMT goes beyond MMT by allowing the data plane to be reconfigured in the following four ways. First, field definitions can be altered and new fields added; second, the number, topology, widths, and depths of match tables can be specified, subject only to an overall resource limit on the number of matched bits; third, new actions may be defined, such as writing new congestion fields; fourth, arbitrarily modified packets can be placed in specified queue(s), for output at any subset of ports, with a queuing discipline specified for each queue. This configuration should be managed by an SDN controller, but we do not define the control protocol in this paper.

The benefits of RMT can be seen by considering new protocols proposed in the last few years, such as PBB [16], VxLAN [22], NVGRE [19], STT [21], and OTV [20]. Each protocol defines new header fields. Without an architecture like RMT, new hardware would be required to match on and process these protocols.

Note that RMT is perfectly compatible with (and even partly implemented by) the current OpenFlow specification. Individual chips can clearly allow an interface to reconfigure the data plane. In fact, some existing chips, driven at least in part by the need to address multiple market segments, already have some flavors of reconfigurability that can be expressed using ad hoc interfaces to the chip.

Many researchers have recognized the need for something akin to RMT and have advocated for it. For example, the IETF ForCES working group developed the definition of a flexible data plane [17]; similarly, the forwarding abstraction working group in ONF has worked on reconfigurability [30]. However, there has been understandable skepticism that the RMT model is implementable at very high speeds. Without a chip to provide an existence proof of RMT, it has seemed fruitless to standardize the reconfiguration interface between the controller and the data plane.

Intuitively, arbitrary reconfigurability at terabit speeds seems an impossible mission. But what restricted form of reconfigurability is feasible at these speeds? Does the restricted reconfigurability cover a large enough fraction of the needs we alluded to earlier? Can one prove feasibility via working silicon that embodies these ideas? How expensive is such an RMT chip compared to a fixed-table MMT chip? These are the questions we address in this paper.

General purpose payload processing is not our goal. SDN/OpenFlow (and our design) aim to identify the essential minimal set of primitives to process headers in hardware. Think of it as a minimal instruction set like RISC, designed to run really fast in heavily pipelined hardware. Our very flexible design is cost-competitive with fixed designs—i.e., flexibility comes at almost no cost.

Paper Contributions: Our paper makes a concrete contribution to the debate of what forwarding abstractions are practical at high speed, and the extent to which a forwarding plane can be reconfigured by the control plane. Specifically, we address the questions above as follows:

1. An architecture for RMT (§2): We describe an RMT switch architecture that allows definition of arbitrary headers and header sequences, arbitrary matching of fields by an arbitrary number of tables, arbitrary writing of packet header fields (but not the packet body), and state update per packet. Several restrictions are introduced to make the architecture realizable. We outline how a desired configuration can be expressed by a parse graph to define headers, and a table flow graph to express the match table topology.

2. Use cases (§3): We provide use cases that show how the RMT model can be configured to implement forwarding using Ethernet and IP headers, and support RCP [7].

3. Chip design and cost (§4–5): We show that the specific form of reconfigurability we advocate is indeed feasible and describe the implementation of a 640 Gb/s (64 × 10 Gb/s) switch chip. Our architecture and implementation study included significant detail in logic and circuit design, floorplanning and layout, using techniques proven over the design team's long history of developing complex digital ICs. An industry standard 28nm process was used. This work is necessary to prove feasibility in meeting goals such as timing and chip area (cost). We have not produced a complete design or actual silicon. Based on our investigation, we show that the cost of reconfiguration is expected to be modest: less than 20% beyond the cost of a fixed (non-reconfigurable) version.

We make no claim that we are the first to advocate reconfigurable matching or that our proposed reconfiguration functionality is the "right" one. We do claim that it is important to begin the conversation by making a concrete definition of the RMT model and showing it is feasible by exhibiting a chip, as we have attempted to do in this paper. While chip design is not normally the province of SIGCOMM, our chip design shows that a rather general form of the RMT model is feasible and inexpensive. We show that the RMT model is not only a good way to think about programming the network, but also lends itself to direct expression in hardware using a configurable pipeline of match tables and action processors.

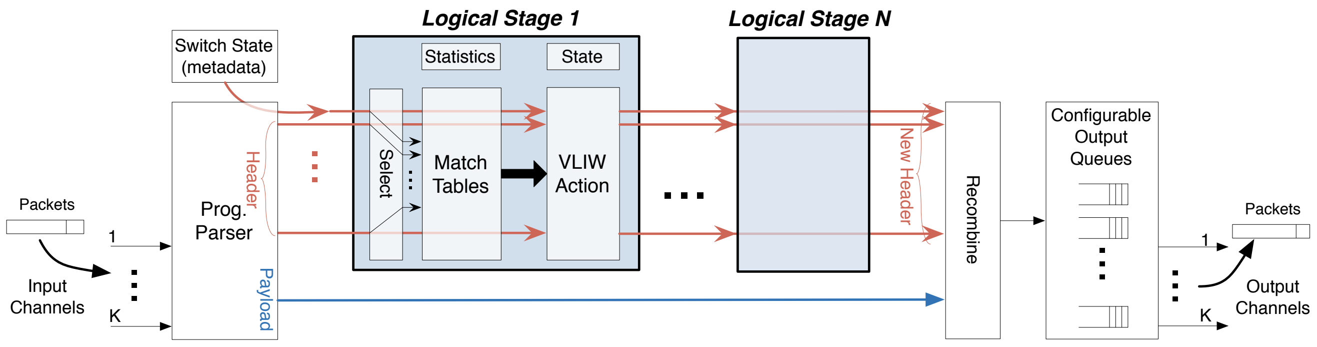
2. RMT ARCHITECTURE

We spoke of RMT as "allow(ing) a set of pipeline stages ... each with a match table of arbitrary depth and width that matches on fields". A logical deduction is that an RMT switch consists of a parser, to enable matching on fields, followed by an arbitrary number of match stages. Prudence suggests that we include some kind of queuing to handle congestion at the outputs.

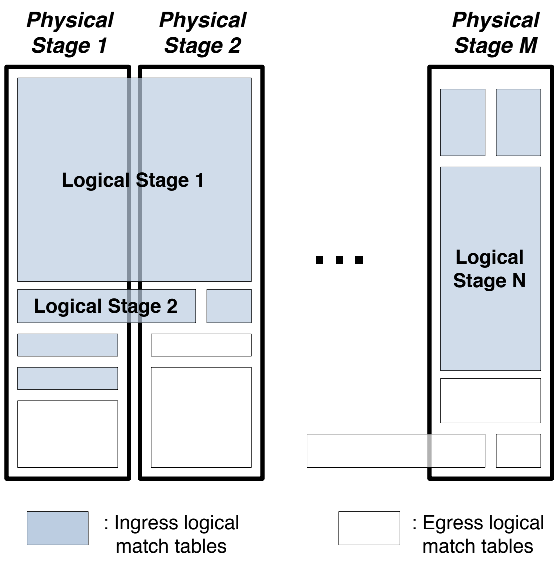
Let's look a little deeper. The parser must allow field definitions to be modified or added, implying a reconfigurable parser. The parser output is a packet header vector, which is a set of header fields such as IP dest, Ethernet dest, etc. In addition, the packet header vector includes "metadata" fields such as the input port on which the packet arrived and other router state variables (e.g., current size of router queues). The vector flows through a sequence of *logical* match stages, each of which abstracts a logical unit of packet processing (e.g., Ethernet or IP processing) in Figure 1a.

Each logical match stage allows the match table size to be configured: for IP forwarding, for example, one might want a match table of 256K 32-bit prefixes and for Ethernet a match table of 64K 48-bit addresses. An input selector picks the fields to be matched upon. Packet modifications are done using a wide instruction (the VLIW—very long instruction word—block in Figure 1c) that can operate on all fields in the header vector concurrently.

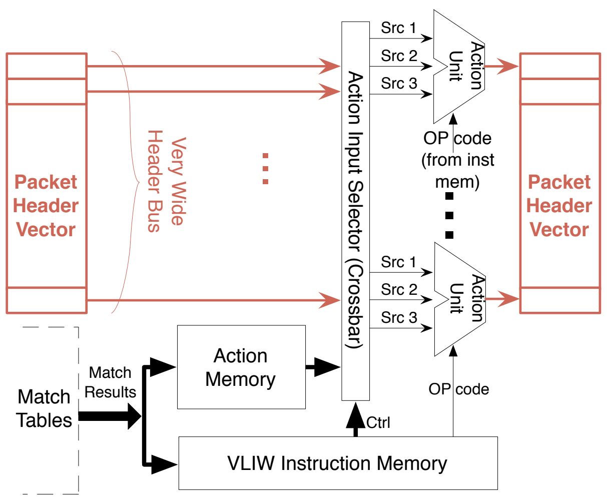
More precisely, there is an action unit for each field F in the header vector (Figure 1c), which can take up



(a) RMT model as a sequence of logical Match-Action stages.



(b) Flexible match table configuration.



(c) VLIW action architecture.

Figure 1: RMT model architecture.

to three input arguments, including fields in the header vector and the action data results of the match, and rewrite F. Allowing each logical stage to rewrite every field may seem like overkill, but it is useful when shifting headers; we show later that action unit costs are small compared to match tables. A logical MPLS stage may pop an MPLS header, shifting subsequent MPLS headers forward, while a logical IP stage may simply decrement TTL. Instructions also allow limited state (e.g., counters) to be modified that may influence the processing of subsequent packets.

Control flow is realized by an additional output, next-table-address, from each table match that provides the index of the next table to execute. For example, a match on a specific Ethertype in Stage 1 could direct later processing stages to do prefix matching on IP (routing), while a different Ethertype could specify exact matching on Ethernet DAs (bridging). A packet's fate is controlled by updating a set of destination ports and queues; this can be used to drop a packet, implement multicast, or apply specified QoS such as a token bucket.

A recombination block is required at the end of the pipeline to push header vector modifications back into the packet (Figure 1a). Finally, the packet is placed in the specified queues at the specified output ports and a configurable queuing discipline applied.

In summary, the ideal RMT of Figure 1a allows new fields to be added by modifying the parser, new fields to be matched by modifying match memories, new actions by modifying stage instructions, and new queueing by modifying the queue discipline for each queue. An ideal RMT can simulate existing devices such as a bridge, a router, or a firewall; and can implement existing protocols, such as MPLS and ECN, and protocols proposed in the literature, such as RCP [7] that uses non-standard congestion fields. Most importantly, it allows future data plane modifications without modifying hardware.

2.1 Implementation Architecture at 640Gb/s

We advocate an implementation architecture shown in Figure 1b that consists of a large number of *physical* pipeline stages that a smaller number of *logical* RMT stages can be mapped to, depending on the resource needs of each logical stage. This implementation architecture is motivated by:

1. Factoring State: Router forwarding typically has several stages (e.g., forwarding, ACL), each of which uses a separate table; combining these into one table produces the cross-product of states. Stages are processed sequentially with dependencies, so a physical pipeline is natural.

2. Flexible Resource Allocation Minimizing Resource Waste: A physical pipeline stage has some resources (e.g., CPU, memory). The resources needed for a logical stage can vary considerably. For example, a firewall may require all ACLs, a core router may require only prefix matches, and an edge router may require some of each. By flexibly allocating physical stages to logical stages, one can reconfigure the pipeline to metamorphose from a firewall to a core router in the field. The number of physical stages N should be large enough so that a logical stage that uses few resource will waste at most $1/N$-th of the resources. Of course, increasing N will increase overhead (wiring, power): in our chip design we chose $N = 32$ as a compromise between reducing resource wastage and hardware overhead.

3. Layout Optimality: As shown in Figure 1b, a logical stage can be assigned more memory by assigning the logical stage to multiple *contiguous* physical stages. An alternate design is to assign each logical stage to a decoupled set of memories via a crossbar [3]. While this design is more flexible (any memory bank can be allocated to any stage), worst case wire delays between a processing stage and memories will grow at least as $\sqrt{M}$, which in router chips that require a large amount of memory M can be large. While these delays can be ameliorated by pipelining, the ultimate challenge in such a design is *wiring:* unless the current match and action widths (1280 bits) are reduced, running so many wires between every stage and every memory may well be impossible.

In sum, the advantage of Figure 1b is that it uses a tiled architecture with short wires whose resources can be reconfigured with minimal waste. We acknowledge two disadvantages. First, having a larger number of physical stages seems to inflate power requirements. Second, this implementation architecture conflates processing and memory allocation. A logical stage that wants more processing must be allocated two physical stages, but then it gets twice as much memory even though it may not need it. In practice, neither issue is significant. Our chip design shows that the power used by the stage processors is at most 10% of the overall power usage. Second, in networking most use cases are dominated by memory use, not processing.

2.2 Restrictions for Realizability

The physical pipeline stage architecture needs restrictions to allow terabit-speed realization:

Match restrictions: The design must contain a fixed number of physical match stages with a fixed set of resources. Our chip design provides 32 physical match stages at *both* ingress and egress. Match-action processing at egress allows more efficient processing of multicast packets by deferring per-port modifications until after buffering.

Packet header limits: The packet header vector containing the fields used for matching and action has to be limited. Our chip design limit is 4Kb (512B) which allows processing fairly complex headers.

Memory restrictions: Every physical match stage contains table memory of identical size. Match tables of arbitrary width and depth are approximated by mapping each logical match stage to multiple physical match stages or fractions thereof (see Fig. 1b). For example, if each physical match stage allows only 1,000 prefix entries, a 2,000 IP logical match table is implemented in two stages (upper-left rectangle of Fig. 1b). Likewise, a small Ethertype match table could occupy a small portion of a match stage's memory.

Hash-based binary match in SRAM is 6× cheaper in area than TCAM ternary match. Both are useful, so we provide a fixed amount of SRAM and TCAM in each stage. Each physical stage contains 106 1K × 112b SRAM blocks, used for 80b wide hash tables (overhead bits are explained later) and to store actions and statistics, and 16 2K × 40b TCAM blocks. Blocks may be used in parallel for wider matches, e.g., a 160b ACL lookup using four blocks. Total memory across the 32 stages is 370 Mb SRAM and 40 Mb TCAM.

Action restrictions: The number and complexity of instructions in each stage must be limited for realizability. In our design, each stage may execute one instruction per field. Instructions are limited to simple arithmetic, logical, and bit manipulation (see §4.3). These actions allow implementation of protocols like RCP [7], but don't allow packet encryption or regular expression processing on the packet body.

Instructions can't implement state machine functionality; they may only modify fields in the packet header vector, update counters in stateful tables, or direct packets to ports/queues. The queuing system provides four levels of hierarchy and 2K queues per port, allowing various combinations of deficit round robin, hierarchical fair queuing, token buckets, and priorities. However, it cannot simulate the sorting required for say WFQ.

In our chip, each stage contains over 200 action units: one for each field in the packet header vector. Over 7,000 action units are contained in the chip, but these consume a small area in comparison to memory ($<$ 10%). The action unit processors are simple, specifically architected to avoid costly to implement instructions, and require less than 100 gates per bit.

How should such an RMT architecture be configured? Two pieces of information are required: a parse graph that expresses permissible header sequences, and a table flow graph that expresses the set of match tables and the control flow between them (see Figure 2 and §4.4). Ideally, a compiler performs the mapping from these graphs to the appropriate switch configuration. We have not yet designed such a compiler.

3. EXAMPLE USE CASES

To give a high level sense of how to use an RMT chip, we will take a look at two use cases.

Example 1: L2/L3 switch. First, we need to configure the parser, the match tables and the action tables. For our first example, Figure 2a shows the parse graph, table flow graph, and memory allocation for our L2/L3 switch. The Parse Graph and Table Flow Graph tell the parser to extract and place four fields (Ethertype, IP DA, L2 SA, L2 DA) on the wide header bus. The Table Flow Graph tells us which fields should be read from the wide header bus and matched in the tables. The Memory Allocation tells us how the four logical tables are mapped to the physical memory stages. In our example, the Ethertype table naturally falls into Stage 1, with the remaining three tables spread across all physical stages to maximize their size. Most hash table RAM is split between L2 SA and DA, with 1.2 million entries for each. We devote the TCAM entries in all 32 stages to hold 1 million IP DA prefixes. Finally, we need to store the VLIW action primitives to be executed following a match (e.g. egress port(s), decrement TTL, rewrite L2 SA/DA). These require 30% of the stage's RAM memory, leaving the rest for L2 SA/DA. If enabled, packet and byte counters would also consume RAM, halving the L2 table sizes.

Once configured, the control plane can start populating each table, for example by adding IP DA forwarding entries.

Example 2: RCP and ACL support. Our second use case adds Rate Control Protocol (RCP) support [7] and an ACL for simple firewalling. RCP minimizes flow completion times by having switches explicitly indicate the fair-share rate to flows, thereby avoiding the need to use TCP slow-start. The fair-share rate is stamped into an RCP header by each switch. Figure 2b shows the new parse graph, table flow graph and memory allocation.

To support RCP, the packet's current rate and estimated RTT fields are extracted and placed in the header vector by the parser. An egress RCP table in stage 32 updates the RCP rate in outgoing packets—the `min` action selects the smaller of the packet's current rate and the link's fair-share rate. (Fair-share rates are calculated periodically by the control plane.)

A stateful table (§4.5) accumulates data required to calculate the fair-share rate. The stateful table is instantiated in stage 32 and accumulates the byte and RTT sums for each destination port.

We also create 20K ACL entries (120b wide) from the last two stages of TCAM, reducing the L3 table to 960K prefixes, along with RAM entries to hold the associated actions (e.g., drop, log).

In practice, the user should not be concerned with the low-level configuration details, and would rely on a compiler to generate the switch configuration from the parse graph and table flow graph.

4. CHIP DESIGN

Thus far, we have used a logical abstraction of an RMT forwarding plane which is convenient for network users. We now describe implementation design details.

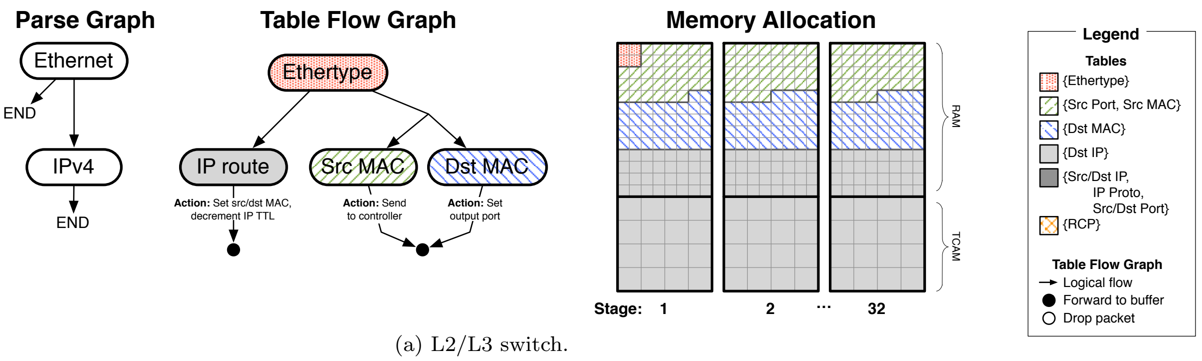
We chose a 1GHz operating frequency for the switch chip because at 64 ports $\times$ 10 Gb/s and an aggregate throughput of 960M packets/s, a single pipeline can process all input port data, serving all ports, whereas at lower frequencies we would have to use multiple such pipelines, at additional area expense. A block diagram of the switch IC is shown in Figure 3. Note that this closely resembles the RMT architectural diagram of Figure 1a.

Input signals are received by 64 channels of 10Gb SerDes (serializer-deserializer) IO modules. 40G channels are made by ganging together groups of four 10G ports. After passing through modules which perform low level signalling and MAC functions like CRC generation/checking, input data is processed by the parsers. We use 16 ingress parser blocks instead of the single logical parser shown in Figure 1a because our programmable parser design can handle 40Gb of bandwidth, either four 10G channels or a single 40G one.

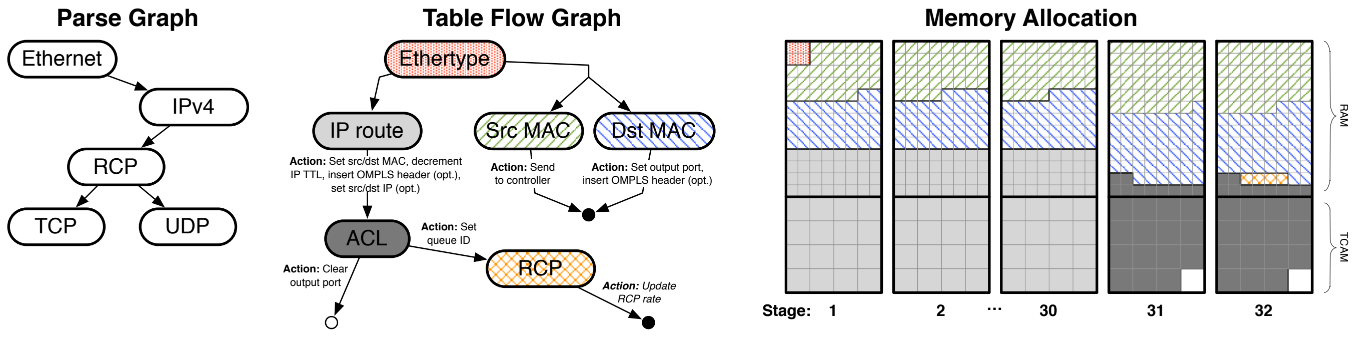
Parsers accept packets where individual fields are in variable locations, and output a fixed 4 Kb packet header vector, where each parsed field is assigned a fixed location. The location is static, but configurable. Multiple copies of fields (e.g., multiple MPLS tags or inner and outer IP fields) are assigned unique locations in the packet header vector.

The input parser results are multiplexed into a single stream to feed the match pipeline, consisting of 32 sequential match stages. A large shared buffer provides storage to accommodate queuing delays due to output port oversubscription; storage is allocated to channels as required. Deparsers recombine data from the packet header vector back into each packet before storage in the common data buffer.

A queuing system is associated with the common data buffer. The data buffer stores packet data, while pointers to that data are kept in 2K queues per port. Each channel in turn requests data from the common data



(a) L2/L3 switch.


Parse Graph
Ethernet
IPv4
RCP
TCP
UDP
Table Flow Graph
Ethertype
IP route
Src MAC
Dst MAC
Action: Set src/dst MAC, decrement IP TTL, insert OMPLS header (opt.), set src/dst IP (opt.)
Action: Send to controller
Action: Set output port, insert OMPLS header (opt.)
ACL
Action: Set queue ID
Action: Clear output port
RCP
Action: Update RCP rate
Memory Allocation
RAM
TCAM
Stage: 1 2 ··· 30 31 32

(b) RCP and ACL support.

Figure 2: Switch configuration examples.

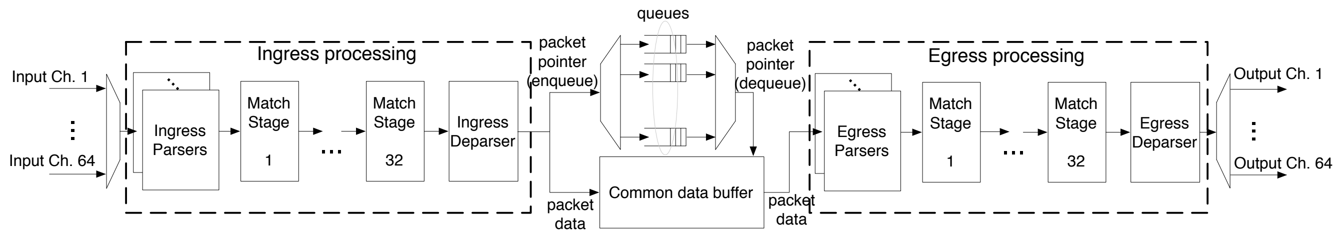


Figure 3: Switch chip architecture.

buffer using a configurable queuing policy. Next is an egress parser, an egress match pipeline consisting of 32 match stages, and a deparser, after which packet data is directed to the appropriate output port and driven off chip by 64 SerDes output channels.

While a separate 32-stage egress processing pipeline seems like overkill, we show that egress and ingress pipelines share the same match tables so the costs are minimal. Further, egress processing allows a multicast packet to be customized (say for its congestion bit or MAC destination) by port without storing several different packet copies in the buffer. We now describe each of the major components in the design.

4.1 Configurable Parser

The parser accepts the incoming packet data and produces the 4K bit packet header vector as its output. Parsing is directed by a user-supplied parse graph (e.g., Figure 2), converted by an offline algorithm into entries in a 256 entry × 40b TCAM, which matches 32b of incoming packet data and 8b of parser state. Note that this parser TCAM is completely separate from the match TCAMs used in each stage. When, for example, the 16-bits of the Ethertype arrives, the CAM matches on 16 of say 32 arriving bits (wildcarding the rest), updating state indicating the next header type (e.g., VLAN or IP) to direct further parsing.

More generally, the result of a TCAM match triggers an action, which updates the parser state, shifts the incoming data a specified number of bytes, and directs the outputting of one or more fields from positions in the input packet to fixed positions in the packet header vector. This loop repeats to parse each packet, as shown in Figure 4. The loop was optimized by pulling critical update data, such as input shift count and next parser state, out of the RAM into TCAM output prioritization logic. The parser's single cycle loop matches fields at 32 Gb/s, translating to a much higher throughput since not all fields need matching by the parser. A 40 Gb/s packet stream is easily supported by a single parser instance.

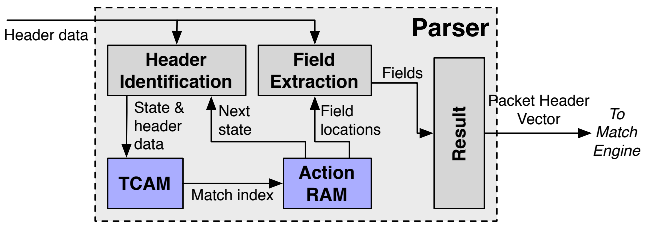


Figure 4: Programmable parser model.

4.2 Configurable Match Memories

Each match stage contains two 640b wide match units, a TCAM for ternary matches, and an SRAM based hash table for exact matches. The bitwidth of the SRAM hash unit is aggregated from eight 80b subunits, while the ternary table is composed of 16 40b TCAM subunits. These subunits can be run separately, in groups for wider widths, or ganged together into deeper tables. An input crossbar supplies match data to each subunit, which selects fields from the 4Kb packet header vector. As described earlier (Figure 1b), tables in adjacent match stages can be combined to make larger tables. In the limit, all 32 stages can create a single table.

Further, the ingress and egress match pipelines of Figure 3 are actually the same physical block, shared at a fine grain between the ingress and egress threads as shown in Figure 1b. To make this work, first the packet header vector is shared between input and output vectors; each field in the vector is configured as being owned by either the ingress or egress thread. Second, the corresponding function units for each field are allocated in the same way to ingress or egress. Lastly, each memory block is allocated to either ingress or egress. No contention issues arise since each field and memory block is owned exclusively by either egress or ingress.

Each match stage has 106 RAM blocks of 1K entries $\times$ 112b. The fraction of the RAM blocks assigned to match, action, and statistics memory is configurable. Exact match tables are implemented as Cuckoo hash tables [9, 26, 32] with (at least) four ways of 1K entries each, each way requiring one RAM block. Reads are deterministically performed in one cycle, with all ways accessed in parallel. Each match stage also has 16 TCAM blocks of 2K entries $\times$ 40b that can be combined to make wider or deeper tables.

Associated with each match table RAM entry is a pointer to action memory and an action size, a pointer to instruction memory, and a next table address. The action memory contains the arguments (e.g., next hop information to be used in output encapsulations), and the instructions specify the function to be performed (e.g., Add Header). Action memory, like the match memory, is made of 8 narrower units consisting of 1K words $\times$ 112 bits, yielding 96b data each (along with field valid bits and memory ECC—error correcting code—bits). Action memory is allocated from the 106 RAM blocks, while action instructions are held in a separate dedicated memory.

As in OpenFlow, our chip stores packet and byte statistics counters for each flow table entry. Full 64b versions of these counters are contained in off-chip DRAM, with limited resolution counters on-chip using the 1K RAM blocks and applying the LR(T) algorithm [33] to provide acceptable DRAM update rates. One word of statistics memory configurably holds the counters for either two or three flow entries, allowing tradeoffs of statistics memory cost vs DRAM update rate. Each counter increment requires a read and write memory operation, but in the 1GHz pipeline only one operation is available per packet, so a second memory port is synthesized by adding one memory bank[1].

4.3 Configurable Action Engine

A separate processing unit is provided for each packet header field (see Figure 1c), so that all may be modified concurrently. There are 64, 96, and 64 words of 8, 16, and 32b respectively in the packet header vector, with an associated valid bit for each. The units of smaller words can be combined to execute a larger field instruction, e.g., two 8b units can merge to operate on their data as a single 16b field. Each VLIW contains individual instruction fields for each field word.

OpenFlow specifies simple actions, such as setting a field to a value, and complex operations, such as PBB encapsulate and inner-to-outer or outer-to-inner TTL copies, where the outer and inner fields may be one of a number of choices. Complex modifications can be subroutines at low speeds, but must be flattened into single-cycle operations at our 1 GHz clock rate using a carefully chosen instruction set.

Table 1 is a subset of our action instructions. Deposit-byte enables depositing an arbitrary field from anywhere in a source word to anywhere in a background word. Rot-mask-merge independently byte rotates two sources, then merges them according to a byte mask, useful in performing IPv6 to IPv4 address translation [18]. Bitmasked-set is useful for selective metadata updates; it requires three sources: the two sources to be merged and a bit mask. Move, like other operators, will only

[1]S. Iyer. Memoir Systems. Private communication, Dec. 2010.

move a source to a destination if the source is valid, i.e., if that field exists in the packet. Another generic optional conditionalization is destination valid. The cond-move and cond-mux instructions are useful for inner-to-outer and outer-to-inner field copies, where inner and outer fields are packet dependent. For example, an inner-to-outer TTL copy to an MPLS tag may take the TTL from an inner MPLS tag if it exists, or else from the IP header. Shift, rotate, and field length values generally come from the instruction. One source operand selects fields from the packet header vector, while the second source selects from either the packet header vector or the action word.

Category	Description
logical	and, or, xor, not, ...
shadd/sub	signed or unsigned shift
arith	inc, dec, min, max
deposit-byte	any length, source & dest offset
rot-mask-merge	IPv4 $\leftrightarrow$ IPv6 translation uses
bitmasked-set	$S_1\&S_2 \mid \overline{S_1}\&S_3$; metadata uses
move	if V_{S_1} $S_1 \rightarrow D$
cond-move	if $\overline{V_{S_2}}\&V_{S_1}$ $S_1 \rightarrow D$
cond-mux	if V_{S_2} $S_2 \rightarrow D$ else if V_{S_1} $S_1 \rightarrow D$

Table 1: Partial action instruction set. (S_i means source i; V_x means x is valid.)

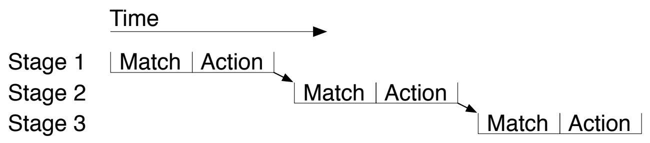
A complex action, such as PBB, GRE, or VXLAN encapsulation, can be compiled into a single VLIW instruction and thereafter considered a primitive. The flexible data plane processing allows operations which would otherwise require implementation with network processors, FPGAs, or software, at much higher cost and power at 640Gb/s.

4.4 Match Stage Dependencies

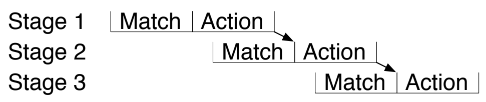
One can easily ensure correctness by requiring that physical match stage I processes a packet header vector P only after stage $I-1$ completely finishes processing P. But this is overkill in many cases and can severely increase latency. Key to reducing latency is to identify three types of dependencies between match tables in successive stages: *match dependencies*, *action dependencies* and *successor dependencies*, each described below.

More precisely, processing in an individual match stage occurs over a number of clock cycles in three phases. Matching occurs, then as a result of a match, actions are taken, with each of those operations requiring several clock cycles. Then finally, the modified packet header vector is output.

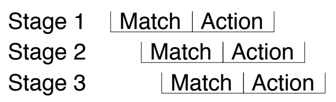
Match dependencies occur when a match stage modifies a packet header field and a subsequent stage matches upon that field. In this case, the first stage must completely finish both match and action processing before the subsequent stage can begin execution. No overlapping in time of the processing of the two match stages is possible, as shown in Figure 5a. A small time gap is shown between the end of first stage execution and the beginning of the second stage execution. This is a transport delay, the time it takes to physically move signals from the output of the first stage to the input of the second stage on chip.



(a) Match dependency.



(b) Action dependency.



(c) No dependency or successor dependency.

Figure 5: Match stage dependencies.

Action dependencies occur when a match stage modifies a packet header field and a subsequent stage uses that field as an input to an action. This differs from the match dependency above in that the modified field is an input to the action processing rather than the earlier match processing. For example, if one stage sets a TTL field and the next stage decrements the TTL, then the result of the first stage is required before executing the action of the second. In this case, partial overlapping of the two match stages' executions is possible, as shown in Figure 5b. The second stage can execute its match, but before its action begins it must have the results from the previous stage. Here, the second stage action begins one transport delay after the first stage execution ends.

Successor dependencies occur when the execution of a match stage is predicated on the result of execution of a prior stage. Each flow match in a table must indicate the next table to be executed, including the default action on a table miss when no flows match; absence of a next table indicates the end of table processing. If A, B, and C are three successive tables, A's execution may specify B as the next table, or alternatively, C or a later table. Only in the first case is table B executed, so B's execution is predicated on the successor indication from A. In this case, the chip runs table B speculatively, and resolves all predication qualifications before any results from B are committed. Predication is resolved inline within the 16 tables of a match stage, and between stages using the inter-stage transport delay. In this case, execution of successive stages is offset only by the transport delay, as shown in Figure 5c. Whereas in this design successor dependencies incur no additional delay, a naïve implementation could introduce as much delay as a match dependency if no speculative table execution is done.

If no dependencies exist between match stages, their execution can be concurrent. Figure 5c applies in this case, where the executions of consecutive stages are offset only by the transport delay.

The pipeline delays between successive stages are statically configurable between the three options of Figure 5, individually for the ingress and egress threads. Pipeline delays are derived by creating and analyzing a table flow graph [30], shown in Figure 6. Each table is represented by a node, with directed edges from each node to all possible successor tables. The graph is annotated with information stating the fields used for matching, the fields modified by each table, and the fields used as action processing inputs of each table. Listing modified output fields separately for each table destination removes false dependencies: a successor table is not dependent on fields modified by alternate successors. Non-local dependencies also have to be identified, e.g., for tables A, B, and C executed in order, if B has no dependencies on A but C does, B may be executed concurrently with A, but C's pipeline delay with respect to A must reflect its dependency. Table typing information can either be (preferably) declared in advance [31, §A.3.5.5] or derived by examining all flow entries. In the absence of any table typing information, no concurrent execution is possible and all match stages must execute sequentially with maximum latency.

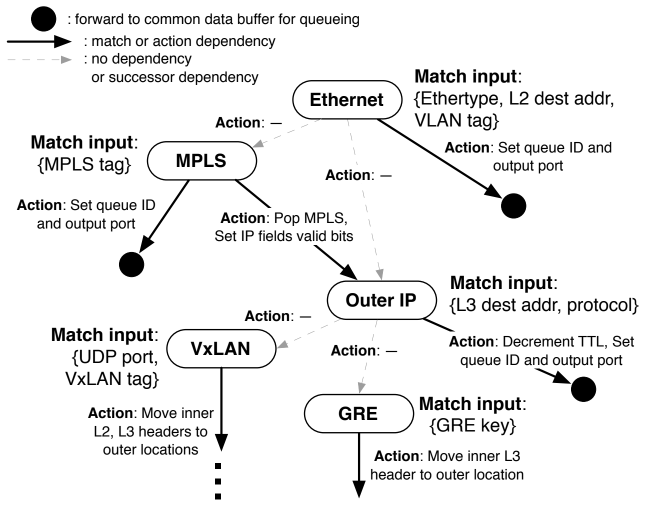

: forward to common data buffer for queueing
: match or action dependency
: no dependency or successor dependency
Ethernet
Match input: {Ethertype, L2 dest addr, VLAN tag}
Action: Set queue ID and output port
Action: –
MPLS
Match input: {MPLS tag}
Action: Set queue ID and output port
Action: Pop MPLS, Set IP fields valid bits
Action: –
Outer IP
Match input: {L3 dest addr, protocol}
Action: Decrement TTL, Set queue ID and output port
Action: –
VxLAN
Match input: {UDP port, VxLAN tag}
Action: Move inner L2, L3 headers to outer locations
Action: –
GRE
Match input: {GRE key}
Action: Move inner L3 header to outer location

Figure 6: Table flow graph.

The pipeline is meant to be static; dependencies between stages for each packet are not analyzed dynamically as in CPU pipelines.

Match dependencies result in 12 cycle latency between match stages, action dependencies result in three cycle latency between stages, and stages with successor dependencies or no dependencies have one cycle between stages.

4.5 Other architectural features

Multicast and ECMP: Multicast processing is split between ingress and egress. Ingress processing writes an output port bit vector field to specify outputs, and optionally, a tag for later matching and the number of copies routed to each port. A single copy of each multicast packet is stored in the data buffer, with multiple pointers placed in the queues. Copies are created when the packet is injected into the egress pipeline, where tables may match on the tag, the output port, and a packet copy count to allow per-port modifications.

ECMP and uECMP processing is similar. Ingress processing writes a bit vector to indicate possible outputs, and an optional weight for each output. The destination is selected when the packet is buffered and the packet is enqueued for a single port. Per-port modifications may be performed in the egress pipeline.

Meters and Stateful Tables: Meters measure and classify flow rates of matching table entries, and can be used to modify or drop packets which exceed set limits. Meter tables are implemented using match stage unit memories provided for match, action, and statistics. Like statistics memories, meter table memories require two accesses per meter operation in a read-modify-write pipeline.

Meters are but one example of *stateful tables*, where an action modifies state that is visible to subsequent packets and can be used to modify them. The design implements a form of stateful counters that can be arbitrarily incremented and reset. Such stateful tables can be used to implement, for example, GRE sequence numbers (that are incremented in each encapsulated packet) and OAM [15, 25]. In OAM, some computation is performed to broadcast packets at prescribed intervals (increment counter) and to raise an alarm if return packets do not arrive by a specified interval (reset counter) and the counter exceeds a threshold.

Consistent and atomic updates: To allow consistent updates [34] (i.e., packets must see either old or the new state, but not a mixture), version information is contained in table entries, and a version ID flows through the pipeline with each packet, qualifying table matches by version compatibility. This also allows atomically updating multiple rules, and associating each packet with a specific table version and configuration, useful for debugging [12].

5. EVALUATION

We characterize the cost of configurability in terms of the increased area and power of our design relative to a conventional less programmable switch chip. Our comparison culminates in a comparison of total chip area and power in Section 5.5. To get there, we consider the contributors to cost by considering the parser, the match stages and the action processing in turn.

5.1 Programmable Parser Costs

Programmability comes at a cost. A conventional parser is optimized for one parse graph, whereas a programmable parser must handle any supported parse graph. Cost is evaluated by comparing synthesis results for conventional and programmable designs. Total gate count is shown in Figure 7 for conventional parsers implementing several parse graphs and a programmable parser. We assume parser aggregate throughput of 640 Gb/s by combining 16 instances of a 40 Gb/s parser running at 1 GHz. The result module in all designs contains the 4Kb packet header vector during parsing. The programmable parser uses a 256 $\times$ 40 bit TCAM and a 256 $\times$ 128 bit action RAM.

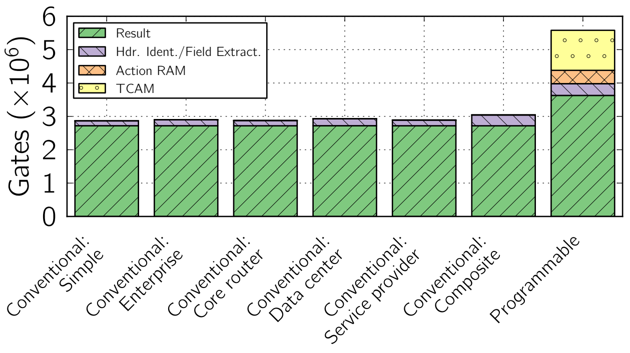


Figure 7: Total gate count of parsers providing 640 Gb/s aggregate throughput.

Parser gate count is dominated by logic for populating the parser header vector. The conventional design requires 2.9–3.0 million gates, depending upon the parse graph, while the programmable design requires 5.6 million gates, of which 1.6 million is contributed by the added TCAM and action RAM modules. From these results, the cost of parser programmability is approximately 2 ($5.6/3.0 = 1.87 \approx 2$).

Despite doubling the parser gate count, the parser accounts for less than 1% of the chip area, so the cost of making the parser programmable is not a concern.

5.2 Memory Costs

There are several costs to memories the reader may be concerned about. First, there is the cost of the memory technology itself (hash table, TCAMs) versus standard SRAM memory and the cost of breaking up the memory into smaller blocks which can be reconfigured; second, there is the cost of additional data needed in each match table entry to specify actions and keep statistics; and third, there is the cost of internal fragmentation such as when an Ethernet Destination address of 48 bits is placed in a 112-bit wide memory. We treat each overhead in turn and specifically point out the (small) additional cost for programmability. In what follows we refer to an entry in a match table (such as a 48-bit Ethernet DA) as a *flow entry*.

5.2.1 Memory Technology Costs

Exact matching: We use cuckoo hashing for exact matching because its fill algorithm provides high occupancy, typically above 95% for 4-way hashtables. Cuckoo hashtables resolve fill conflicts by recursively evicting conflicting entries to other locations. Additionally, while our memory system is built up out of 1K by 112 bit RAM blocks for configurability, one might expect an area penalty vs using larger, more efficient memory units. However, using 1K RAM blocks incurs an area penalty of only about 14% relative to the densest SRAM modules available for this technology.

Wildcard matching: We use large amounts of TCAM on chip to directly support wildcard matching such as prefix matching and ACLs. TCAM is traditionally thought to be infeasible due to power and area concerns. However, TCAM operating power has been reduced by about 5$\times$ by newer TCAM circuit design techniques [1]. Thus, in the worst case, at the maximum packet rate with minimum packet size on all channels, TCAM power is one of a handful of major contributors to total chip power; at more typical mixtures of long and short packets, TCAM power reduces to a small percentage of the total.

Next, while a TCAM typically has an area 6–7$\times$ that of an equivalent bitcount SRAM, both ternary and binary flow entries have other bits associated with them, including action memory, statistics counters, and instruction, action data, and next table pointers. For example, with 32 bits of IP prefix, 48 bits of statistics counter, and 16 bits of action memory (say for specifying next hops), the TCAM portion is only 1/3 of the memory bitcount and so the area penalty for TCAM drops to around 3$\times$.

While 3$\times$ is significant, given that 32b (IPv4) or 128b (IPv6) longest prefix matching and ACLs are major use cases in all existing routers, devoting significant resources to TCAM to allow say 1M IPv4 prefixes or 300K ACLs seems useful. While we could have used just SRAM with special purpose LPM algorithms instead as in [5], achieving the single-cycle latency of TCAMs for a 32 or 128 bit LPM is difficult or impossible. Nevertheless, deciding the ratio of ternary to binary table capacity (our chip proposes a 1:2 ratio) is an important implementation decision with significant cost implications, for which currently there is little real world feedback.

5.2.2 *Costs of Action Specification*

Besides the flow entry, each match table RAM entry also has a pointer to action memory (13b), an action size (5b), a pointer to instruction memory (5b for 32 instructions), and a next table address (9b). These extra bits represent approximately 35% overhead for the narrowest flow entries. There are also bits for version and error correction but these are common to any match table design so we ignore them.

In addition to overhead bits in a flow entry, other memories are required for storing actions and statistics. These add to the total overhead—the ratio of total bits required to just the match field bits— but both of these extra costs can sometimes be reduced. We will show how in some cases it is possible to reduce flow entry overhead bits. Furthermore, applications require varying amounts of action memory, and sometimes statistics are not needed, so these memory costs can be reduced or eliminated.

Given the variable configuration of memory blocks between match, action, and statistics, we use a few configuration examples to see how the bookkeeping overhead varies compared to non-configurable fixed allocations.

In the first configuration, shown in Figure 8a and Table 2, 32 memory blocks are used for match memory in a stage, implementing 32K 80b wide exact match flow entries. Another 16K 80b ternary entries are in the TCAM modules. All flow entries have action entries of the same size, requiring 48 memories for actions. Statistics consume 24 memory banks, along with a spare bank for multiporting the statistics memory. An approximately equal portion of action memory is allocated for each match memory, and might be considered a base case with a minimum amount of flow table capacity.

Excluding the 24 banks used for ternary actions and statistics in case a, 40% of the banks used for binary operations are match tables, indicating a 2.5× overhead. Compounding this with the 35% bit overhead in the match tables, the total binary overhead is 3.375×, the ratio of total bitcount to match data bitcount. In other words, only a third of the RAM bits can be used for flow entries.

Cases a_2 and a_3 of Table 2 change the match width to 160 and 320 bits respectively, reducing action and statistics requirements, and yielding increased match capacity.

Configuration b of Table 2 further increases the binary and ternary flow table match widths to 640 bits, as shown in Figure 8b, (memory width is shown horizontally), reducing the number of flow entries by a factor of 8 compared to the base case, along with the required action and statistics capacity. While such a wide match may be rare (say for an entire header match), we see that with 8× wider flow entries than the base case above, 80 banks can be used for exact match, 75% of memory capacity, 2.5× higher table capacity than the base case.

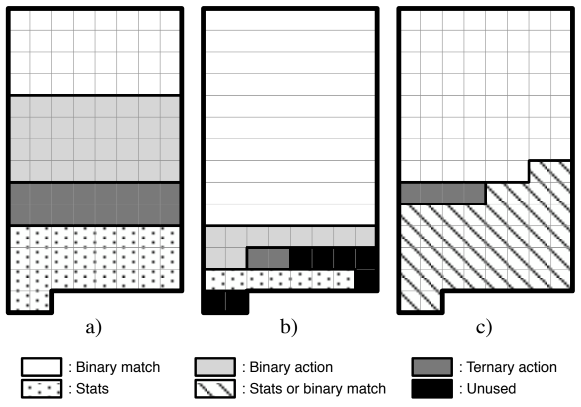


Figure 8: Match stage unit memory map examples.

Configuration c_1 of table 2, shown in Figure 8c, takes advantage of a use case where the number of individual actions is limited. For example, a data center address virtualization application requires a large number of flow entries, but match entries may point to one of only 1000 possible destination top of rack switches. Thus 4K of action memory would suffice. That would allow 62 memory blocks for match and 40 for statistics, almost doubling the number of exact match entries from the base case. If statistics were not desired, 102 memory blocks could be used for match as shown in table entry c_2, 96% of the total memory capacity.

Case	MatchWidth	Match	Action	Stats	Relative
a_1	80	32	48	25	1.000×
a_2	160	52	34	18	1.625×
a_3	320	72	22	12	2.250×
b	640	80	12	7	2.500×
c_1	80	62	4	40	1.900×
c_2	80	102	4	0	3.250×

Table 2: Memory unit allocation and relative exact match capacity.

In short, flow entry density is greatly increased if actions or statistics are reduced or eliminated. If the user of the chip deems these complex actions and statistics necessary, then it is unfair to blame the chip configurability options for this overhead. The only fundamental book-keeping costs that can be directly attributed to programmability are the instruction pointer (5b) and the next table address (9b), which is around 15%.

Costs can be further reduced in tables with fixed behaviors. A fixed-function table uses the same instruction and next-table pointers for all entries, e.g., L2 dest MAC table; static values may be configured for these attributes, allowing 14 bits of instruction and next-table pointer to be reclaimed for match. More generally, a configurable width field in the flow entry can optionally provide LSBs for action, instruction, or next-table addresses, allowing a reduced number of different instructions or actions, or addressing a small array for next table, while reclaiming as many instruction, next-table, and action address bits as possible given the complexity of the function.

Next, tables can provide an action value as an immediate constant rather than as a pointer to action memory for small constants, saving pointer and action memory bits.

A simple mechanism enables these optimizations: match table field field boundaries can be flexibly configured, allowing a range of table configurations with arbitrary sizes for each field, subject to a total bitwidth constraint. Tables with fixed or almost fixed functions can be efficiently implemented with almost no penalty compared to their fixed counterparts.

5.2.3 *Crossbar Costs*

A crossbar within each stage selects the match table inputs from the header vector. A total of 1280 output bits (640b for each of TCAM and hash table) are selected from the 4Kb input vector. Each output bit is driven by a 224 input multiplexor, made from a binary tree of and-or-invert AOI22 gates (with the logic function $\overline{AB + CD}$), and costing one $0.65\mu m^2$ per mux input. Total crossbar area is $1280 \times 224 \times 0.65\mu m^2 \times 32$ stages $\approx 6\,mm^2$. Area computation for the action unit data input muxes is similar.

5.3 Fragmentation Costs

A final overhead is internal fragmentation or packing costs. Clearly, a 48-bit Ethernet Destination Address placed in an 112 b wide memory wastes more than half the memory. In comparison, a fixed-function Ethernet bridge contains custom 48-bit wide RAM. Thus this cost is squarely attributable to programmability and our choice of 112b wide RAMs. One could reduce this overhead for Ethernet by choosing 48b as the base RAM width, but how can a chip designed for general purpose use (and future protocols) predict future match identifier widths?

Fortunately, even this overhead is reduced through another architectural trick that allows *sets of flow entries to be packed together* without impairing the match function. For example, the standard TCP 5-tuple is 104 bits wide. Three of these entries can be packed into four memory units of width 448 b, rather than separately requiring each to consume two memory units. Or, with low entry overhead equivalent to a simple pass/fail, 4 of these can be packed into 4 words, due to amortization of ECC bits over wider data. Fundamentally, this is possible efficiently because under the covers, longer matches are constructed from trees of smaller 8-bit matches; flexibility only slightly complicates this logic.

The combination of these two techniques, variable data packing into a data word (to reduce action specification costs) and variable flow entry packing into multiple data words (to reduce fragmentation costs), assures efficient memory utilization over a wide range of configurations. In summary, while conventional switches have highly efficient implementations for specific configurations of tables, this architecture can approach that efficiency, not only for those specific table configurations, but for a wide range of other configurations as well.

5.4 Costs of Action Programmability

Besides the costs of the instruction RAMs and action memories, there are around 7000 processor datapaths ranging in width from 8 to 32 bits. Fortunately, because they use a simple RISC instruction set, their combined area consumes only 7% of the chip.

5.5 Area and Power Costs

This switch is designed with large match table capacity, so match and action memories contribute substantially to chip area estimates as shown in Table 3. The first item, which includes IO, data buffer, CPU, etc., occupies a similar area in conventional switches. As can be seen, the VLIW action engine and parser/deparser contributions to area are relatively small.

We suggested earlier that the match stage unit RAMs suffered a 14% area penalty compared to the best possible RAMs. Given this penalty to the match stage SRAM (not TCAM) area, and some allowance for additional bitcount vs conventional switches (15%), excess memory area is about 8% of the chip total. If excess logic in the parser and action engine add another 6.2%, a 14.2% area cost results, justifying the earlier claim of a less than 15% cost differential.

Section	Area	Cost
IO, buffer, queue, CPU, etc	37.0%	0.0%
Match memory & logic	54.3%	8.0%
VLIW action engine	7.4%	5.5%
Parser + deparser	1.3%	0.7%
	Total extra cost:	**14.2%**

Table 3: Estimated chip area profile.

Estimated switch power is detailed in Table 4 under worst case operating conditions (temperature, chip process), 100% traffic with a mix of half min and half max (1.5 kB) size packets, and all match and action tables

Section	Power	Cost
I/O	26.0%	0.0%
Memory leakage	43.7%	4.0%
Logic leakage	7.3%	2.5%
RAM active	2.7%	0.4%
TCAM active	3.5%	0.0%
Logic active	16.8%	5.5%
	Total extra cost:	**12.4%**

Table 4: Estimated chip power profile.

filled to capacity. Input/output power is equivalent to a conventional switch. Memory leakage power is proportional to memory bitcount, so if this programmable switch can be implemented with equivalent bitcount to a conventional switch, power will be comparable. The remaining items, totalling 30%, will be less in a conventional switch because of the reduced functionality in its match-action pipeline. We estimate that our programmable chip dissipates 12.4% more power than a conventional switch, but is performing much more substantial packet manipulation.

The overall competitive evaluation with conventional switches suggests that equivalent functions can be performed by this switch with equivalent memory bitcounts. This in turn drives parity in dominant aspects of chip cost and power. The additional power and area costs borne by the programmable solution are quite small given the more comprehensive functionality of the switch.

6. RELATED WORK

Flexible processing is achievable via many mechanisms. Software running on a processor is a common choice. Our design performance exceeds that of CPUs by two orders of magnitude [6], and GPUs and NPUs by one order [4,8,11,29].

Modern FPGAs, such as the Xilinx Virtex-7 [35], can forward traffic at nearly 1 Tb/s. Unfortunately, FPGAs offer lower total memory capacity, simulate TCAMs poorly, consume more power, and are significantly more expensive. The largest Virtex-7 device available today, the Virtex-7 690T, offers 62Mb of total memory which is roughly 10% of our chip capacity. The TCAMs from just two match stages would consume the majority of lookup-up tables (LUTs) that are used to implement user-logic. The volume list price exceeds $10,000, which is an order of magnitude above the expected price of our chip. These factors together rule out FPGAs as a solution.

Related to NPUs is PLUG [5], which provides a number of general processing cores, paired with memories and routing resources. Processing is decomposed into data flow graphs, and the flow graph is distributed across the chip. PLUG focuses mainly on implementing lookups, and not on parsing or packet editing.

The Intel FM6000 64 port $\times$ 10Gb/s switch chip [24] contains a programmable parser built from 32 stages with a TCAM inside each stage. It also includes a two-stage match-action engine, with each stage containing 12 blocks of 1K $\times$ 36b TCAM. This represents a small fraction of total table capacity, with other tables in a fixed pipeline.

The latest OpenFlow [31] specification provides an MMT abstraction and partly implements an RMT model. But its action capability is still limited, and it is not certain that a standard for functionally complete actions is on the way or even possible.

7. CONCLUSIONS

Ideally, a switch or router should last for many years. Dealing with a changing world requires *programmability* that allows software upgrades to add new functions and new protocols in the field. Network processors (NPUs) were introduced to support this vision, but neither NPUs or GPUs have come close to achieving the speeds of fixed function switches using ASICs; nor have we seen a case study of reprogramming an NPU-based router such as Cisco's CRS-1 to add a new protocol. Likewise FPGAs, which only recently approached ASIC forwarding speeds, remain prohibitively expensive.

Our chip design resurrects this ancient vision of programmability, expressed in the RMT model, within the constraints of what is possible on a real chip. New fields can be added, lookup tables can be reconfigured, new header processing added, all through software reconfiguration. While our chip cannot do regular expressions, or manipulate packet bodies, a box built from this chip could metamorphose from an Ethernet chip on Tuesday to a firewall on Wednesday and to a completely new device on Thursday, all by the right software upgrades. The challenge is to do this today at a capacity approaching a terabit. The chip design we propose has surprising specs: it contains 7,000 processor datapaths, 370 Mb of SRAM, and 40 Mb of TCAM, across 32 processing stages.

In terms of ideas, we single out the RMT model as a powerful way to map the programmer's desired forwarding behavior onto a pipeline built from a flexible parser, a configurable arrangement of logical match stages with memories of arbitrary width and depth, and flexible packet editing. These abstractions require new algorithms to be efficiently implemented at terabit speeds. Our use of memory blocks that can be ganged within or across stages is key to realizing the vision of reconfigurable match tables; our large scale use of TCAM greatly increases matching flexibility; and finally, our use of fully-parallel VLIW instructions is key to packet editing. Our design suggests that this greatly increased flexibility comes at an additional cost of less than 15%

in area and power consumption of the chip. Ultimately, the implementation challenge which we have addressed that may not be apparent to SIGCOMM audiences is producing an architecture which is possible to wire efficiently on chip.

While the OpenFlow specification hints at RMT and several researchers [17] have actively pursued this dream, RMT models remained theoretical without an existence proof of a chip design that works at terabit speeds. Our paper contributes a concrete RMT proposal and a proof of its feasibility. Clearly it is possible to go further and remove some of the restrictions we have imposed for implementation reasons. But now the conversation can begin.

Acknowledgements

We would like to thank Nick Feamster for feedback on an early draft, Aditya Akella for shepherding the final version, and the anonymous reviewers for their useful comments.

8. REFERENCES

[1] P. Bosshart. Low power ternary content-addressable memory (TCAM). US Patent 8,125,810, Feb. 2012.

[2] Brocade. Software-Defined Networking. http://www.brocade.com/launch/sdn/index.html.

[3] F. Chung, R. Graham, J. Mao, and G. Varghese. Parallelism versus memory allocation in pipelined router forwarding engines. *Theory of Computing Systems*, 39(6):829–849, 2006.

[4] Cisco. QuantumFlow Processor. http://newsroom.cisco.com/dlls/2008/hd_030408b.html.

[5] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. PLUG: flexible lookup modules for rapid deployment of new protocols in high-speed routers. *SIGCOMM Comput. Commun. Rev.*, 39(4):207–218, Aug. 2009.

[6] 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. In *Proc. SOSP '09*, pages 15–28, 2009.

[7] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, 2008.

[8] EZchip. NP-5 Network Processor. http://www.ezchip.com/p_np5.htm.

[9] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38:229–248, 2005.

[10] J. Fu and J. Rexford. Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. In *Proc. ACM CoNEXT '08*, pages 1–12, 2008.

[11] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, Aug. 2010.

[12] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. HotSDN '12*, pages 55–60, 2012.

[13] U. Hölzle. OpenFlow @ Google. In *Open Networking Summit*, April 2012. http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf.

[14] HP. OpenFlow – Software-Defined Network (SDN). http://www.hp.com/OpenFlow/.

[15] *IEEE Std 802.1ag-2007: Amendment 5: Connectivity Fault Management*, pages 1–260. 2007.

[16] *IEEE Std 802.1ah-2008: Amendment 7: Provider Backbone Bridges*, pages 1–110. 2008.

[17] IETF. *RFC 5810 Forwarding and Control Element Separation (ForCES) Protocol Specification*, March 2010.

[18] IETF. *RFC 6052 IPv6 Addressing of IPv4/IPv6 Translators*, October 2010.

[19] IETF. *NVGRE: Network Virtualization using Generic Routing Encapsulation*, Feb. 2013. https://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-02.

[20] IETF. *Overlay Transport Virtualization*, Feb. 2013. https://tools.ietf.org/html/draft-hasmit-otv-04.

[21] IETF. *A Stateless Transport Tunneling Protocol for Network Virtualization (STT)*, Mar. 2013. https://tools.ietf.org/html/draft-davie-stt-03.

[22] IETF. *VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*, May 2013. https://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-04.

[23] Indigo – Open Source OpenFlow Switches. http://www.openflowhub.org/display/Indigo/Indigo+-+Open+Source+OpenFlow+Switches.

[24] Intel Ethernet Switch Silicon FM6000. http://ark.intel.com/products/series/64370.

[25] ITU-T. *OAM Functions and Mechanisms for Ethernet Based Networks G.8013/Y.1731*, July 2011.

[26] A. Kirsch, M. Mitzenmacher, and U. Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.*, 39(4):1543–1561, Dec. 2009.

[27] N. McKeown, T. Anderson, H. Balakrishnan,

G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[28] NEC. ProgrammableFlow Networking. http://www.necam.com/SDN/.

[29] Netronome. NFP-6xxx Flow Processor. http://www.netronome.com/pages/flow-processors/.

[30] Open Networking Foundation. *Fowarding Abstractions Working Group*. https://www.opennetworking.org/working-groups/forwarding-abstractions.

[31] Open Networking Foundation. *OpenFlow Switch Specification*. Version 1.3.1.

[32] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Journal of Algorithms*, pages 122–144, 2004.

[33] S. Ramabhadran and G. Varghese. Efficient implementation of a statistics counter architecture. In *Proc. SIGMETRICS '03*, pages 261–271, 2003.

[34] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *SIGCOMM Comput. Commun. Rev.*, 42(4):323–334, Aug. 2012.

[35] Xilinx. 7 series FPGA overview. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.