Rethinking Packet Forwarding Hardware

Martin Casado[*§] Teemu Koponen[†] Daekyeong Moon[‡] Scott Shenker[‡§]

1 Introduction

For routers and switches to handle ever-increasing bandwidth requirements, the packet "fast-path" must be handled with specialized hardware. There have been two approaches to building such packet forwarding hardware. The first is to embed particular algorithms in hardware; this is what most commodity forwarding chips do (*e.g.*, those from Broadcom, Marvell, and Fulcrum). These chips have led to amazing increases in performance and reductions in cost; for instance, one can now get 24 ports of gigabit ethernet for under $1000.

Unfortunately, this approach offers only very rigid functionality; one can't change protocols or add new features that require hardware acceleration without redoing the chip. This forces network forwarding enhancements to evolve on hardware design timescales, which are glacially slow compared to the rate at which network applications and requirements are changing.

To counter this inflexibility, several vendors have taken a different approach by introducing more flexible "network processors". These have not been as successful as anticipated, for at least two reasons. First, designers were never able to find a sweet-spot in the tradeoff between hardware simplicity and flexible functionality, so the performance/price ratios have lagged well behind commodity networking chips. For another, the interface provided to software has proven hard to use, requiring protocol implementors to painstakingly contort their code to the idiosyncrasies of the particular underlying hardware to achieve reasonable performance. These two problems (among others) have prevented general-purpose network processors from dislodging the more narrowly targeted commodity packet forwarding hardware that dominates the market today.

In this paper, we step back from these two well-known approaches and ask more fundamentally: what would we want from packet forwarding hardware, how might we achieve it, and what burden does that place on networking software?

Ideally, hardware implementations of the forwarding paths should have the following properties:

- *Clean interface between hardware and software:* The hardware-software interface should allow each to evolve independently. Protocol implementations should not be tied to particular networking hardware, and networking hardware should not be restricted to particular protocols. General-purpose network processors have largely failed the first requirement, while the current generation of commodity networking chips fail the second requirement.
- *Hardware simplicity:* In order to scale to increasing speeds, the hardware functionality should be of very limited complexity. Again, both current approaches to hardware acceleration fail to satisfy this requirement.
- *Flexible and efficient functionality:* The resulting software-hardware combination should be capable of realizing a wide variety of networking functions at high-speed and low-cost, while having short development cycles. Commodity chips work at high-speed and low-cost, but have long development cycles for new functionality. General-purpose network processors are not yet competitive on speed or cost.

In this paper we propose a different approach to hardware packet forwarding that has the potential to realize all of these goals. Our approach assigns a completely different role to hardware. Traditionally, hardware implementations have embodied the logic required for packet forwarding. That is, the hardware had to capture all the complexity inherent in a packet forwarding decision.

In contrast, in our approach all forwarding decisions are done first in software, and then the hardware merely *mimics* these decisions for subsequent packets to which that decision applies (*e.g.*, all packets destined for the same prefix). Thus, the hardware does not need to understand the logic of packet forwarding, it merely caches the results of previous forwarding decisions (taken by software) and applies them to packets with the same headers. The key task is to match incoming packets to previous decisions, so the required hardware is nothing more than a glorified TCAM, which is simple to implement (compared to other networking hardware) and simple to reason about. We describe this approach in more detail in Section 2.

One key challenge in this approach is scaling to high speeds. Achieving high forwarding rates is not a matter of the number of clock cycles needed to make a forwarding decision, but instead is one of cache hit rates; what fraction of packet forwarding can be done based on the forwarding of previous packets? This question is central to the viability of this approach, and

[*]Nicira Networks Inc.
[†]Helsinki Institute for Information Technology (HIIT)
[‡]UC Berkeley, Computer Science Division
[§]International Computer Science Institute (ICSI)

we address it in Section 3. Another key challenge is deciding to which packets a particular forwarding decision applies (*i.e.*, which fields in the packet header must match in order to apply a decision), and when this decision has been rendered obsolete by changing network state (*e.g.*, changes in forwarding tables); these system design issues are addressed in Section 4.

This approach has been motivated by a long line of flow-oriented approaches to networking hardware, from [5] to [2] (and many others). What makes our approach different from traditional connection-oriented architectures (*e.g.*, ATM) is that our approach does not require any change to networking protocols. Our approach (similar to the references [2, 5]) only changes the interface between hardware and software, but does not affect protocols or the Internet architecture. Like [3] we seek to provide a flexible software model with hardware forwarding speeds, however in our approach we propose to fully decouple the software from the hardware, while [3] provides direct access to hardware resources such as the TCAM and special purpose ASICs. We view this as a necessary and complimentary step for integrating with existing hardware.

2 Our Approach

2.1 General Description

Packet forwarding decisions deterministically depend on the header of the arriving packet and on some local state in the router/switch (such as a routing table).[1] For example, the header/state combination for ethernet forwarding is the destination MAC address and the L2 learning table, whereas for IP forwarding it is the destination IP address and the FIB.

The forwarding decision includes both the output port(s) on the network device as well as possible modifications to the packet header, such as the label swapping in MPLS or packet encapsulation/decapsulation. As long as the relevant pieces of local state have not changed, then all packets with the same packet header should receive the same forwarding behavior. This statement applies to *any* forwarding protocol that depends only on packet headers and local state.

To leverage this fact, we treat packet forwarding as a *matching process*, with all packets matching a previous decision handled by the hardware, and all non-matching packets handled by the software. We assume that when a packet is handled by software, we can infer a *flow entry* for that packet which contains four pieces of information:

Ingress port(s): This specifies to which ingress ports a flow entry applies. Many rules will only apply to the ingress port of the packet that generated the flow entry, but other rules (such as "packets with TTL=1 should be dropped") can be applied to all ingress ports.

Matching rule: The software specifies the fields in the packet header that must match for the flow entry to apply. This could be the entire header (*i.e.*, requiring a perfect match), or only some fields (*e.g.*, perhaps only the destination address to match).

Forwarding action: The forwarding action is the set of output ports to which the packet should be sent, along with (optionally) the rewritten packet headers for each port. Dropping the packet is represented by a special null output port.

State dependence: This represents the local state on which the forwarding decision is based (such as entries in a routing table or ACL database).

When a packet arrives, it is compared to the table of flow entries corresponding to its ingress port. If its packet header does not match any of the entries, then it is sent to software for processing. If it matches a flow entry, the packet is handled according to the specification of the forwarding action in the flow entry. Whenever a piece of local state changes, such as an update to an ACL or routing table, then all flow entries that depended on that state are invalidated and removed from the flow table.

Before proceeding to examples, we mention a few more detailed points. First, the matching rules must be defined so a packet matches no more than one rule. Since we assume that the software deterministically forwards each packet based on its header fields and particular system state, it should be possible to generate non-conflicting entries at runtime.

Second, we view stateful tunneling (*e.g.*, IPsec in which a sequence number is updated for every packet) as virtual output ports rather than part of the forwarding logic. This is similar to how they are handled in standard software IP forwarding engines in which they are modeled as networking interfaces. If we did not use this technique then no flow entry would be recorded for such flows, because the state on which the forwarding actions depends (which includes per-packet modifications) would be updated on every new packet arrival.

Third, there are a number of possible system designs that could determine the headers and system state used in a decision, such as a special-purpose programming language, a specialized compiler or even hardware support. We discuss these further in Section 4. However, the primary goal of this paper is not to give a specific and comprehensive system design (we are working on this and plan to discuss it in future work), but rather to explore the fundamental viability of our general approach.

2.2 Examples

To demonstrate how our proposal operates in practice, we discuss two examples of familiar datapath functions in

[1] We are ignoring forwarding decisions based on deep-packet inspection.

Packet fields	Memory dependencies	Output	Modifications
eth_type = VLAN, VLAN ID	VLAN database, port configuration, L2 MAC table	access port	remove VLAN
eth_type ! = VLAN	VLAN database, port configuration, L2 MAC table	access port	None
eth_type = VLAN, VLAN ID	VLAN database, port configuration, L2 MAC table	trunk port	None
eth_type ! = VLAN	VLAN database, port configuration, L2 MAC table	trunk port	add VLAN

1: Packet fields, memory dependencies and potential packet modifications for VLAN tagging example.

```
1  if eth.type ≠ IPv4 then
2      send_to(drop port); return;
3  end
4  if ip.ttl ≤ 1 then
5      generate_icmp(); return;
6  end
7  if not valid ip.chksum then
8      send_to(drop port); return;
9  end
10 decrement(ip.ttl); update(ip.chksum);
11 if no ip.options and not ip.fragmented then
12     next_hop, dst_port ⟵ fib(ip.dst_ip);
13     eth.src_mac ⟵ dst_port.mac;
14     eth.dst_mac ⟵ arp_cache(next_hop);
15     send_to(dst_port);
16 else
17     // Complex header processing...
18 end
```

1: Pseudo code for IP routing.

heavy use today, IPv4 routing and VLAN tagging.

IP routing. We begin by describing how our approach operates with standard IPv4 routing (pseudo-code shown in Figure 1). From the standpoint of caching, IPv4 is non-trivial in that it includes multiple state dependencies as well as a number of header modifications (TTL decrement, checksum update, L2 address updates).

On packet receipt, the router performs basic header checks (protocol version, TTL), and verifies the checksum before doing a prefix lookup in the FIB. Hence, the forwarding decision depends on all of the protocol fields as well as the locally stored FIB. Moreover, because the checksum is dependent on every byte in the header (except itself), every header field must be included in the flow entry (including the TTL).[2] However, as we show in the next section, even with the full header included in the matching rule, our approach can achieve very high cache hit rates.

After looking up the next hop, the router consults the ARP cache for the next hop MAC address. The Ethernet header is updated with the MAC of the outgoing interface as well as the next hop MAC address before being sent out. Thus, local port MAC configuration, and ARP cache are added as state dependencies for the flow entry.

VLAN tagging. We consider a simple VLAN tagging implementation, which implements port-based VLAN encapsulation and decapsulation for packets on access ports, L2 forwarding within VLANs, and pass-through of VLAN packets on trunk ports. Table 1 summarizes the packet header and memory dependencies of this example for given inputs. Briefly, if an incoming packet was received on an access port, it is tagged with the VLAN ID associated with that port. This adds the state dependency of the VLAN database to any resulting decision. Once the associated VLAN has been identified (if any) the packet VLAN ID and destination MAC addresses are used for L2 forwarding. This adds the MAC learning tables to the state dependency of the flow entry. In the final step, the packet is sent out of one or more ports as determined by the L2 forwarding step. If the port is an access port (and the packet has a VLAN header), the VLAN header is removed. This adds the port configuration as a dependency.

Tagging and label swapping (*e.g.*, MPLS in addition to VLAN) fit nicely with our proposal as the associated configuration state generally changes very slowly. However, for large networks, MAC learning can generate additional overhead due to learning table entry timeouts. To validate that these are manageable in practice, we explore the cache hit and miss ratios of L2 learning over enterprise traces in the following section.

The two examples presented in this section are limited to lookup and forwarding. However, additional datapath mechanisms, such as ACLs, can be added by simply updating the software. In the case of ACLs, this would merely add the ACL table to the state dependencies. Adding forwarding for additional protocols is similarly straightforward. Since the protocol type is implicitly included in the matching rule, it will automatically be de-multiplexed by the hardware.

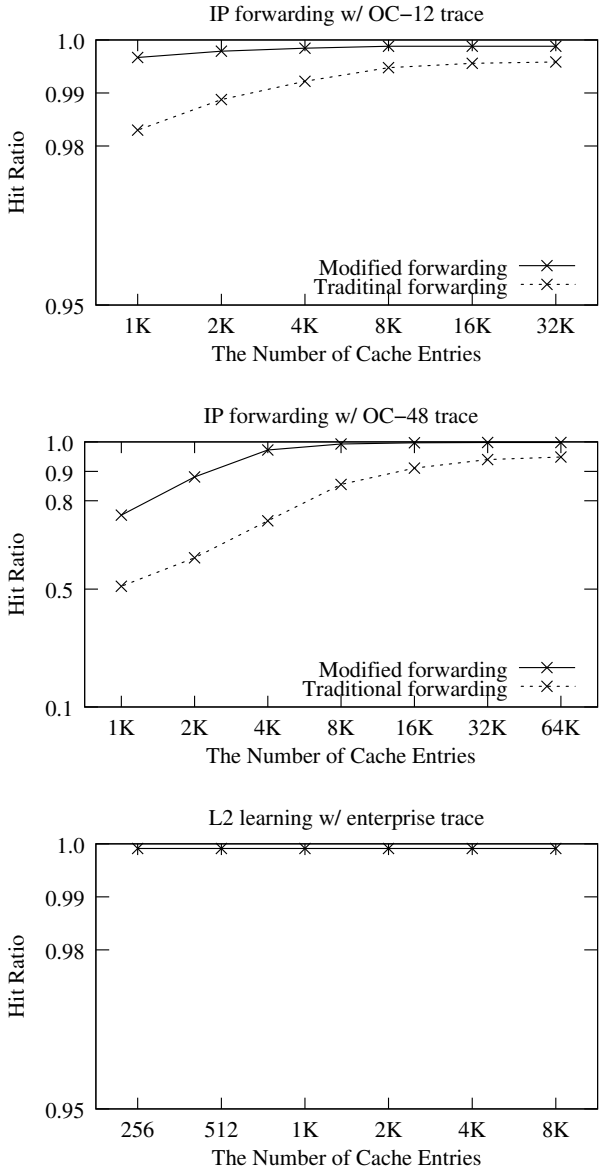
3 Scaling

In order for the software not to become a bottleneck, the cache hit rates must correspond to the speed differential between the software and hardware forwarding rates. For example, a commodity 48x1Gigabit networking chip has roughly two orders of magnitude greater forwarding capacity than a software solution using a standard PC architecture (*e.g.*, [11] or [8]). Therefore, to maintain hardware-only speeds for an integrated architecture with

[2]This can be avoided by relying on endpoint checksum verification as is used in IPv6 and has been suggested elsewhere [7].

similar performance characteristics, the hit rates must exceed 99%.

In this section, we analyze cache hit rates using standard L2 and L3 forwarding algorithms on network traces. We explore the performance of our approach as described here, and also with minor modifications that help achieve vastly better hit rates.



2: Results of cache hit-rate simulations for L2 and L3 forwarding algorithms.

IP forwarding. We first explore the cache behaviour of standard IPv4 forwarding. For this analysis we use two publicly available traces [9, 10], one collected from an OC-48 link consisting of 6,795,675 packets covering a 5 minute time window, and a second collected from an OC-12 link consisting of 6,872,338 packets covering one hour.

We assume the naive hardware implementation of a managed TCAM in which each matching rule fills up an entry. The system uses LRU when replacing cache entries and we assume that the FIB does not change during the period of analysis.

We test two variants of IPv4 fowarding and show the results in Figure 2. The first algorithm (*traditional forwarding* in the graph) strictly obeys the IPv4 forwarding specification (including full header checksum), and thus, requires a full packet header match to be cached. Reaching a 99% hit rate requires a TCAM of 4,096 entries in the OC-12 case. For the OC-48 trace, the results are significantly worse; the hit rate only reaches 95% even with an essentially infinite number of entries.

In the second variant (*modified forwarding* in the graph), we make two modifications which greatly improve the results. First, we perform an incremental update of the checksum which is limited to the TTL decrement change. As a result, no header values beyond the version, TTL, and destination are included in the cache entry.[3] Secondly, the most precise lookup is a /24 rather than full IP match (this is not an unreasonable assumption for high bandwidth links; most FIBs on such links won't contain prefixes smaller than /24s). These changes greatly improve the hit rates. In the case of the OC 48, a 99% hit rate is achieved at 8,000 entries. Further, the lowest hit rate we recorded using the OC-12 traces was 99.9889%, sufficient for speed differentials of over *three* orders of magnitude.

L2 learning. We also look at the performance of decision caching for standard L2 learning. Our MAC learning implementation uses 15 second timeouts of learned addresses and the hardware uses an LRU cache replacement strategy. For analysis, we use publicly available enterprise trace data [1] covering a 3 hour time period, and containing 17,472,976 packets with 1958 unique MAC addresses.

The cache hit rates of this analysis are shown in the bottom most graph of Figure 2. For all entry sizes we tested (the minimum being 256) we found the cache hit rate to be over 99.9%. We note that the use of soft state in this manner appears to have significantly lower TCAM requirements than today's commercial Ethernet switches which commonly hold 1 million Ethernet addresses (6MB of TCAM).

While much more study is warranted on the scaling properties of caching, we feel that the results are promising. Commodity TCAMs with entry widths suitable for L2 and L3 decision caching (*e.g.*, 36 bytes) and with 8k entries (sufficient for all but the OC-48 case) cost about $50 in bulk at the time of this writing. Further, we

[3]We note that IPv6 does not require per-hop IP checksums, as the design community found them redundant.

note that we make modest assumptions as to the relative forwarding speeds of the on-board CPUs. With the onset of multicore, the prospects for the speed differential between the CPU and the hardware forwarding decreasing are good. As long as the forwarding doesn't involve per-packet state changes, the software should scale as a function of the number of cores.[4]

4 System Design Approaches

If the basic matching approach we are advocating can achieve sufficiently high speeds, as we discussed in the last section, then the next challenge is whether we can determine the correct matching rules and state dependencies. There are two extreme approaches one can take (though we suspect an intermediate compromise will be adopted in practice): explicit definition (by the implementor), or automated inference via runtime analysis (*e.g.*, dynamic binary instrumentation). We discuss these in turn.

4.1 Explicit Dependency Identification

The simplest approach to determining matching rules and state dependencies is to put the onus of identifying them on the programmer. This could be done manually, forcing the programmer to explicitly annotate the forwarding decisions with the headers and system state it relies on. Note that while the programmer has to explicitly define these dependencies, they do not have to explicitly manage the flow entries themselves; once the matching rules and state dependencies are specified, a compiler would then generate the necessary code to invalidate any related flow entries upon updates to local state.

Similar methods have been used for global state management in parallel programming environments (see *e.g.*, [6]), and should be straightforward to apply to packet forwarding. An alternative approach to extending the compiler is to provide the developer with a set of library calls that provide similar functionality to the annotations.

While conceptually simple, this explicit identification approach imposes a burden on the programmer and increases the risk of error; even if the algorithm is implemented correctly, a missing state dependency could result in incorrect runtime behavior. Further, it requires that *all* state dependencies be identified, which complicates the use of non-annotated third-party libraries.

4.2 Transparent Dependency Identification

It would be far easier for programmers if they could focus on merely implementing the required forwarding decisions, and let the system automatically infer what the matching rules and state dependencies were. Unfortunately, while forwarding logic itself may be simple, deducing the headers and state that effect decisions is difficult without developer cooperation. For example, some of the challenges are:

- Most forwarding software will have state not directly used as a part of the forwarding decision. This includes counters, file handles, and any configuration state. In an extreme case, a global timer value is used to timestamp all incoming packets. In a naive implementation, every time-tick would invalidate all flow entries. Complex data structures further complicate the analysis by maintaining their own internal state which may change without impacting the outcome of a forwarding decision.
- Processor architecture may require the forwarding logic to access state that isn't part of the resulting decision. For example, prefixes for LPM are commonly read as a single word (beginning from the least significant bit), while only the most significant bits may be relevant to the decision.
- Pipelined and parallel execution models require careful dependency management to attribute a given state access with a particular packet.

Despite these challenges, it is worth considering whether it is possible to determine the headers and state dependencies via runtime analysis by using, for example, dynamic binary instrumentation. This could dramatically increase software overheads, so we are not yet convinced of its practicality, but we discuss it here as a promising avenue for future research.

Runtime analysis operates by tracking *all* memory references of the forwarding software while processing packets. This can be done at the granularity of bits [4], which would be optimal for our application. A simple heuristic would be to assume that any header value that was accessed by the software, or any static or heap state, is a dependency.

Clearly this approach requires disciplined programming over a limited programming model and could not effectively be applied to existing software. Further, the inference is only transparent at the syntax level. The developer must be aware of the runtime characteristics of the system and act accordingly (for example by avoiding spurious reads to global data).

4.3 Inferring Packet Modifications

In addition to inferring important headers and state dependencies, the system must also determine the actions to apply to the packet. While it may be possible to transparently infer the changes at runtime using the techniques discussed in the previous section, a simpler approach would have the developer specify actions explicitly in a manner similar to [2]. To be sufficiently general, the action set must be able to support the modification of arbitrary bytes as well

[4] Although, admittedly, the bus speeds between the CPUs and network hardware require special attention.

as byte insertion (for encapsulation) and deletion (for decapsulation).

5 Conclusions

In all but the lowest end switches and routers, packet forwarding is largely done in hardware. While not often a subject of study in the academic literature, the advances in packet forwarding hardware has been remarkable. In fact, commodity networking chips now support aggregate speeds that only a few years ago were only available on the highest-end routers.

As successful as this generation of hardware-accelerated packet forwarding has been, in the years ahead it must find a way to accommodate two trends that appear to be on a collision course:

- Speed: the demand for bandwidth continues to grow, in enterprises, datacenters, and the wide-area Internet. Backbone links have transitioned from 20Gbps to 40Gbps, core switches in datacenters have high densities of 10Gbps ports, and ever-faster switches and routers are on the horizon.
- Control: the need for better network management and security, particularly in the areas of traffic engineering and access control, has increased emphasis on measures for controlling packet forwarding.

Dealing with increasing speed in hardware calls for limiting the complexity of forwarding decisions (so they can be done efficiently in hardware) and limiting the flexibility of these decisions (so the hardware does not need to be changed). On the other hand, attaining greater control over forwarding decisions calls for greater complexity in the forwarding path, and for greater flexibility (since the nature of these control decisions will change far more frequently than the basic protocols change).

The approach described here tries to accommodate these conflicting desires by retaining full generality of function while remaining simple to implement (by hardware designers) and use (by software implementors). Any forwarding function that depends only on the packet header and local state can be implemented over the same hardware, with a very straightforward interface.

Achieving this generality and ease-of-use at high speeds requires a large enough TCAM-like cache to achieve a very low cache miss rate. Thus, the viability of our approach depends on future trends in hardware (which determines the cost of a given cache size and the speed of software processing) and network traffic (which determines the necessary cache size for a given cache hit rate). We can't make definitive projections about any of these, but our initial investigations suggest that our approach may indeed be viable. In particular, if we focus on IPv6, with its lack of per-hop checksum, then the required cache sizes are very inexpensive.

Of course, this is all very preliminary, and we hope to soon embark on a much fuller investigation. This will entail a more extensive set of traces, a more thorough analysis of the factors that determine the cache miss rate, and building a fully functioning implementation of this approach.

6 References

[1] LBNL/ICSI Enterprise Tracing Project. `http://www.icir.org/enterprise-tracing`.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communications Review*, 38(2):69–74, 2008.

[3] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma. API Design Challenges for Open Router Platforms on Proprietary Hardware . In *ACM HOTNETS '08*, Oct. 2008.

[4] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of ACM SIGPLAN Programming Language Design and Implementation (PLDI '07)*, pages 89–100, 2007.

[5] P. Newman, G. Minshall, and T. L. Lyon. IP Switching - ATM under IP. *IEEE/ACM Transactions on Networking*, 6(2):117–129, 1998.

[6] The OpenMP API Specification for Parallel Programming. `http://openmp.org/wp/`.

[7] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, and G. D. Troxel. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, 1998.

[8] Portwell NAR-5520 Switching Appliance. `http://www.portwell.com/products/detail.asp?CUSTCHAR1=NAR-5522`.

[9] C. Shannon, E. Aben, kc claffy, and D. Andersen. The CAIDA Anonymized 2007 Internet Traces - Jan 2007. `http://www.caida.org/data/passive/passive_2007_dataset.xml`.

[10] C. Shannon, E. Aben, kc claffy, D. Andersen, and N. Brownlee. The CAIDA OC48 Traces Dataset - April 2003. `http://www.caida.org/data/passive/passive_oc48_dataset.xml`.

[11] Vyatta Dell860. `http://www.vyatta.com/products/vyatta_appliance_datasheet.pdf`.