Bridging the Software/Hardware Forwarding Divide

Daekyeong Moon[*] *Jad Naous*[†] *Junda Liu*[*] *Kyriakos Zarifis*[§] *Martin Casado*[‡]
Teemu Koponen[‡] *Scott Shenker*[§*] *Lee Breslau*[¶]

1 Introduction

As network speeds increase, and network requirements diversify, networking hardware vendors are being pressed on two distinct fronts. To remain competitive, they must continually improve their cost/performance ratio at a pace faster than Moore's Law. Over the past fifteen years, enterprise network speeds have increased by three orders of magnitude, from 10Mbps to 10Gbps, requiring that the cost of a unit of bandwidth (a gigabit port, or Gport) drop precipitously. At the same time, networks are expected to address myriad application, management and security requirements that were unheard of fifteen years ago, creating pressure to develop network forwarding devices that can flexibly accommodate these new demands as they arise. This trend is reflected in the many "open" platform initiatives launched by system vendors [10, 16, 22].

Despite significant research and engineering efforts, the industry has failed to simultaneously achieve low cost/performance and high flexibility.[1] There are three basic approaches to building network forwarding devices currently pursued in the field and, as we relate below, they offer quite different tradeoffs between these two goals.

In the commercial world, where cost concerns dominate, the most prevalent approach to building network forwarding devices (this includes both switches and routers, but hereafter we will use the term "switch" as shorthand) involves embedding the basic forwarding logic in ASICs. There are commodity forwarding chips (from, for example, Broadcom, Marvell, and Fulcrum) that cost as little as \$20/10Gport, a price-point that would have been unthinkable only a few years ago. Multi-layer switches using these commodity chips support 240Gbps (24x10Gbps) of capacity for under \$100/10Gport, and 640Gbps (64x10Gbps) chipsets will be generally available this year which should lower the cost even further. However, one cannot modify the hardware forwarding algorithm without respinning the chip. This forces packet forwarding to evolve on hardware design timescales, which are glacially slow compared to the rate at which network requirements are changing.

In the research community, where flexibility is paramount, there has been renewed interest in purely software-based switches (*i.e.*, switches running commodity operating systems on general-purpose CPUs); see, for example, [11]. These designs have the requisite flexibility, since new forwarding algorithms merely require additional programming, but their port-density and cost/performance remains poor. For example, software routers on general CPUs have been shown to support 10Gbps of trivial forwarding for minimum size packets [11], achieving roughly \$1000/10Gport, an order of magnitude more than the ASIC-based switches. While prevalent in low-speed wireless devices, pure software switches have made almost no penetration into the high-speed, high-fanout wireline market.

The industry has tried to bridge the divide between these two extremes with "network processors", whose goal was to provide programmable functionality at near-commodity hardware costs. Unfortunately, network processors have not realized this goal, as designers have yet to find a sweet-spot in the tradeoff between hardware simplicity and flexible functionality. The cost/performance ratios have lagged well behind commodity networking chips and 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.[2] These two problems (among others) have prevented network processors from dislodging the more narrowly targeted commodity packet-forwarding hardware that dominates the market today. One area in which network processors are widely used is in network appliances, which go beyond mere forwarding to do deep packet inspection; but even here, after initial success, network processors are starting to lose market share as the forwarding performance of general-purpose CPUs has significantly improved. Today many market-leading middleboxes (such as load balancers) are implemented on standard x86 platforms.

In this paper, we try again to bridge the software/hardware divide, proposing a hybrid design

[*]UC Berkeley, [†]Stanford University, [‡]Nicira Networks, [§]International Computer Science Institute (ICSI), [¶]AT&T Research

[1]As we clarify later, we are not addressing network "appliances" that provide deep packet inspection. Our concern here is *only* for packet forwarding decisions: to which output port does this packet go, and how should its header be rewritten? When we refer to flexibility, we mean the flexibility in forwarding algorithms, not general programmability for sophisticated packet processing.

[2]However, CAFE [20] is a promising approach for datacenter forwarding.

that attempts to combine the high speed and low cost of hardware with the superior flexibility of software. We motivate our approach by noting that current hardware-accelerated packet-forwarding implements the forwarding logic in hardware. In contrast, in our approach all forwarding decisions are first made in software and thereafter *imitated* by hardware. The hardware uses a classification engine to match software-made decisions with the incoming packets to which they apply (*e.g.*, all packets destined for the same prefix). Thus, the hardware does not need to understand the logic of packet forwarding, it merely stores the results of forwarding decisions (previously made by software) and applies them to packets with the appropriate headers. In short, hardware caches the decisions made by software and executes them at high-speed.

Similarly, the forwarding software is not tied to a particular hardware implementation. Instead, forwarding algorithms are programmed against a high-level API, which increases portability and reduces the complexity of implementation. This forwarding software is not speed-critical, so it can be written in a high-level language (in our implementations we have used Python).

For lack of a better term, we call this decision-caching approach *Software-Defined Forwarding* (SDF), since all forwarding decisions are made in software. To demonstrate the viability of SDF, we built a complete system in hardware and software. On top of it, we have implemented a number of conventional forwarding algorithms (*e.g.*, L2 Ethernet forwarding and L3 LPM forwarding), ported an existing code base (XORP), as well as implemented several recently proposed algorithms (SEATTLE, *i3*, Chord). We have also implemented a virtualization layer in software that allows multiple forwarding algorithms to operate in parallel on the same hardware; doing so did not require any change to the hardware forwarding model.

It is important at this point to clarify the relationship between this work and ongoing work on OpenFlow [21] and NOX [14].[3] The goal of OpenFlow is to provide an API to the hardware that is sufficiently general for network innovation. The goal of NOX is to build a centralized network operating system on top of this hardware interface. Our project lies somewhere in between.

In contrast to OpenFlow, we are not focusing on the definition of the hardware interface, but instead are investigating how to build the hardware *and* software for a router-like system in which the forwarding logic is insulated from the hardware. This requires us to focus on a high-level API instead of just the low-level hardware interface. We envision OpenFlow as a natural candidate for the underlying hardware interface to our higher-layer software forwarding layer, but don't limit ourselves to that choice since our point is more general.

In contrast to the NOX, we are not advocating a centralized management paradigm, but only whether single-box routers and switches could be built in a far more flexible manner.[4] To this end, we provide trace-based performance numbers of this approach, which can be seen as evidence that an OpenFlow-like hardware interface provides sufficient performance for a variety of today's forwarding tasks.

This approach was previously discussed in [8], but the study provided here is far more detailed and comprehensive. Our approach also has similarities with flow caching, and we explain the difference shortly. Because our approach uses packet classification as a basic primitive, we rely on the vast body of work in this area.

2 Goals, Limitations, and Applicability

Before delving into design details and performance results, we first articulate SDF's goals, limitations and applicability.

Goals: The immediate goal of the SDF approach is to build switches with more forwarding flexibility by having all forwarding decisions determined completely in software and then imitated by hardware. This will allow networks to deploy new forwarding algorithms (such as to support a new protocol) without changing their networking hardware. In fact, as we discuss later, this will enable networks to run several different forwarding algorithms in parallel on the same switch.

The longer term goal is to create an environment where both hardware and software designers can independently focus on their respective tasks. Hardware designers can focus on building "decision caches" with higher speeds, greater fanouts, larger memory, lower power and lower cost. Similarly, software designers can address emerging networking requirements by implementing new forwarding algorithms against a simple and fixed hardware interface. This strict separation of concerns should engender rapid progress in both hardware and software.

Limitations: We reiterate that the SDF approach only supports standard header-based forwarding and does not apply to more complex networking behaviors such as deep-packet inspection or arbitrary packet modification (like encryption).

More specifically, the forwarding decisions can depend only on the header of the incoming packet (where the

[3]This discussion also applies to Orphal [22]; there are important technical differences between Orphal and OpenFlow but they are not relevant for our discussion here.

[4]The NOX approach is sometimes referred to as Software-Defined Networking; our use of the term Software-Defined Forwarding emphasizes this distinction between our narrow focus on forwarding in a single box, and NOX's emphasis on global network abstractions.

definition of "header" is the first n bits of the packet, where n is flexible) and on the internal state of the forwarding logic implementation. The output packets can differ from the incoming packet in the content (and the length) of the header, but this new header content can *not* depend on any forwarding logic state modified per packet (that is, for all packets matching a particular entry, the headers must be modified in the same way). Thus, the model does not include, for example, en/de-capsulation, en/de-cryption, nor complex modifications to the packet payload. Such operations requiring unique modifications per packet are handled outside the model, by a logical output port to which the packet is forwarded to. We note this is consistent with the model commonly used in routers and switches of using logical interfaces for tunneling (*e.g.*, IPsec or GRE).

While the packet processing model is limited, we have been surprised by how broadly it applies to traditional and newly proposed forwarding paradigms. For example, the SDF approach can implement aggregation, basic tagging, address overwriting (*e.g.*, VLAN, or layered addressing), dynamic learning and filtering, all without changes to the hardware forwarding model.

Applicability: The SDF approach is most relevant in settings that need a high degree of flexibility. Research is an obvious arena where it would be important to deploy new forwarding algorithms, running at high speeds on commercial-grade equipment, merely by rewriting the forwarding software. In addition, the ability to run several forwarding algorithms in parallel, as is envisioned in GENI and other testbeds, would be very valuable.

It is an open question whether production networks need this degree of flexibility. Some have predicted that the future of networking lies in the ability to virtualize low-level packet transport, in which case this flexibility would be crucial. But it is also true that current forwarding algorithms evolve slowly, which suggests that perhaps flexibility is not relevant to production networks. The question is whether this slow evolution reflects the inability to evolve more quickly or the lack of a need to. We don't pretend to know the answer.

Since SDF, as described so far, relies on "decision-caching", it would be natural to assume that its applicability is limited to regimes where caching is effective. However, as we describe later, SDF can also be run in a "proactive" mode, where all software forwarding decisions are made proactively and stored in hardware. In this case, SDF functions at hardware speeds.

There are some cases where, in order to reduce the state required to cache decisions, it would be advantageous to augment the hardware with a few basic computational primitives — such as decrement by 1, compute a checksum, or compute a hash — that can be applied to a specified set of bits. These primitives are trivial to implement in hardware, and would only be required in cases where performance was critical (*e.g.*, Internet backbone). We discuss this more fully in Section 4.

3 Overview

In this section we present a high-level overview of our approach, leaving design details for Section 6 and a description of our implementation for Section 7. Below we first discuss the system functionality and components, then walk through the steps in forwarding a packet.

3.1 System Functionality and Components

Figure 1 provides a high-level view of the proposed forwarding architecture: it contains three components: the system software, the forwarding hardware, and the forwarding software.

In SDF, a forwarding device accepts an *(inport, inpacket)* pair, and then outputs one or more *(outport, outpacket)* pairs. The hardware performs packet header lookup, header rewriting, and forwarding (passing the packet from the input port to the appropriate output port). In our implementation, a central component of the hardware is a wide TCAM, which performs packet classification based on the packet header (*e.g.*, the first 576 bits of the packet); and, if a match is found, the hardware sends the packet to the specified output port(s) after modifying the packet headers. The hardware also maintains hit counters for TCAM entries but it does not manage the entries without instructions from the system software; in particular, it does not implement inactivity timeouts of any sort.

Forwarding logic (in the form of a forwarding application) is built on the API provided by the system software. The system software is responsible for translating the forwarding application's decisions into entries in the packet classification engine (a TCAM in our design); it is also responsible for revoking these entries when the relevant internal forwarding application state changes. More specifically, for a forwarding decision the system records *a)* the incoming port, *b)* the relevant header bits of the incoming packet (*i.e.*, which header bit should match and what the bit values should be in order for the decision to apply), *c)* the outgoing ports, and *d)* the associated

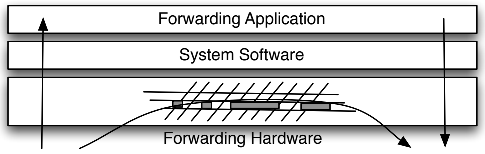


Figure 1: Packets can take one of two paths through the system. A packet matching an entry in the hardware is immediately forwarded out the appropriate port(s). A received packet without a matching entry in the hardware is passed to the forwarding application, which makes a forwarding decision and sends the packet out. The system software then stores this decision in the hardware.

outgoing packet headers. It also records what forwarding application internal state the decision was based on, so it can revoke the classification engine entry corresponding to the decision when the internal state changes. For example, with L3 forwarding, the forwarding decision depends on the matching (software) FIB entry, and if that changes the TCAM entry would be revoked.

The forwarding application is written in a high-level such as C, Java, or even Python. Our goal is to provide a programming model which approaches that of a pure software router. However, due to the difficulty in inferring state dependencies precisely and efficiently, in our current implementation the forwarding application has the responsibility to explicitly mark which header bits and internal state it relied on in a forwarding decision. We discuss the API in more detail in Section 6.

3.2 Forwarding Steps

We now step through a simplified IP forwarding decision to illustrate hardware state setup and revocation. For clarity we ignore TTL decrement and header checksum recomputation (and discuss them in Section 4).

1. A packet is received on $port_0$.
2. Packet classification hardware checks the incoming port and the packet header against its lookup table. Assuming there is no matching entry, the hardware forwards the packet to the system software which passes it on to the forwarding application.
3. The forwarding application checks for the Ethernet type and reads the destination IP address.
4. The forwarding application looks into its software-FIB and finds a matching entry.
5. The forwarding application reads the destination MAC from the ARP cache and overwrites the destination MAC in the packet header with it.
6. The forwarding app sends the packet out of $port_1$.
7. The system software intercepts the packet, saving the new packet header with any changes it may have.
8. The system software creates a hardware packet classification entry which matches on the Ethernet type, and the destination IP address. It creates and associates with the entry an action of overwriting the header with the new header (which has a modified destination MAC), and the action of sending the packet out of $port_1$.
9. The system software also maintains a mapping from the dependent software-FIB and ARP entries to the new hardware forwarding entry.
10. Assuming a second packet is received to the same destination IP address (whether or not from the same transport connection), the hardware will match on the Ethernet type and destination IP address, overwrite the header with the correct destination MAC, and forward it out of $port_1$.
11. If the FIB or ARP entries are modified (*e.g.*, routing update or timeout), the system software removes the associated forwarding entry from the hardware.

3.3 Sound Familiar?

Superficially, SDF seems much like flow or route caching, a technique popular a decade ago (see [17] for a recent revival of the concept). Roughly, flow caching expects the software to use the first packet of every network flow to make a complex forwarding decision (such as IP lookup using LPM) after which the flow is added to a flow cache. The cache entry is only valid for the duration of that flow. However, due to the small average flow sizes (*e.g.*, 10 packets), hit rates tend not to be sufficient for overcoming the difference between hardware and software processing speeds (*i.e.*, the switch is limited by the rate at which software can forward, because it must handle roughly one out of every ten packets). Our approach differs from the classic flow and route caching in three essential ways.

First, we don't couple hardware state to network *flows*, but to forwarding *decisions* (which usually apply to many flows and do not time out, but instead are explicitly revoked as needed). Thus, the forwarding hardware experiences vastly lower miss rates, similar to other soft state commonly found in networks today such as L2 learning tables and ARP caches. We explore the performance of decision caching in Section 5.

Second, flow caching is generally tied to a particular forwarding mechanism (traditionally LPM) and was not designed to increase the *flexibility* of the system, merely to reduce state requirements. In contrast, our approach is intended to support arbitrary forwarding logic over arbitrary headers with the same simple hardware.

Finally, the SDF approach is not limited to reactively populating the hardware forwarding state after a cache miss, as is done in route/flow caching. As we describe later in Section 6, our proposed implementation supports *proactively* pushing decisions into the hardware before any cache miss occurs. Doing so avoids all caching-related performance issues and allows the system to forward at hardware speeds. Thus, while we believe SDF is capable of running reactively in many production deployment environments, it can also be used in environments where caching is not applicable.

To relate our approach to OpenFlow, note that OpenFlow defines a low-level protocol interface to the hardware while SDF defines a high-level programming API. Forwarding applications should be written to the SDF interface, and in turn SDF could use OpenFlow as an interface to manage the switch hardware, or could use the chip vendors native SDK. Writing forwarding applications directly to OpenFlow would limit them to OpenFlow-based switches and require them to manually manage the flow table: insert entries, keep track of state changes, and revoke entries as needed.

4 Example Use Cases

The goal of SDF is to allow software to define a wide range of forwarding behaviors, all of which can then be accelerated by the same underlying hardware. To illustrate the kinds of forwarding that can be supported, below we describe several example use cases.

4.1 Accelerating Existing Software Routers

We first briefly explain how existing software routers such as XORP and Quagga can integrate with SDF. Software routers typically consist of a control plane (which implements a set of routing/switching protocols) that exports the resulting FIB to a forwarding plane such as XORP's Forwarding Engine Abstraction. To port a software router to SDF, it is sufficient to provide the control plane with a forwarding plane implementation built on the APIs SDF provides (which we describe in Section 6). This SDF-based forwarding plane implementation stores the FIB it receives from the control plane and makes forwarding decisions when invoked by the SDF system software. As long as the software router has a well-defined forwarding engine abstraction, implementing the forwarding engine based on SDF's APIs is easy.

To demonstrate the simplicity of the task, we modified XORP's FEA to push its FIB entries over TCP to a Python forwarding application built on top of the SDF APIs. In our prototype implementation, adding the required TCP integration mostly involved copy-pasting from the existing FEA source code, while the forwarding application required only 220 lines of Python.

Thus, we think SDF offers an extremely easy path to provide existing software routers hardware acceleration.

4.2 Virtualization

There is a growing research literature (see [1, 3, 26] for a small sampling) on network virtualization, which is the ability to run separate network architectures in parallel each within their own "slice" of the network. These slices can implement their own routing and forwarding algorithms, and essentially operate as independent networks.[5] To create slices, the traffic must be partitioned in some way so that it is clear which packets belong to which slices. This demultiplexing criterion can be defined in terms of input ports, VLANs or other packet header fields.

[5]We are using the term network virtualization as it is primarily used in the research community. In the commercial world, network virtualization does not involve different forwarding algorithms but is mostly focused on sharing physical infrastructure, managing logical link topology, and supporting virtual server mobility.

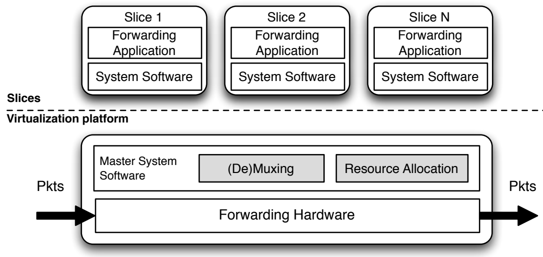


Figure 2: Supporting network virtualization on SDF.

The key challenge in network virtualization is to support separate forwarding algorithms on the same hardware infrastructure. Note that it is simple to do so in *software* by having separate forwarding processes. When a packet arrives at a software switch, it is sent to the appropriate forwarding process based on the demultiplexing criterion, and that process then makes the forwarding decision for that packet.

Initial attempts at producing the same virtualization behavior in hardware were cumbersome and expensive, involving separate network processors for each slice. This is because traditional hardware-based packet forwarding embeds the forwarding logic directly in the hardware, and it is difficult to implement several different forwarding logics simultaneously in the same ASIC.

However, because SDF uses software for all forwarding decisions, and these decisions are merely imitated in hardware, supporting multiple forwarding algorithms is straightforward (as long as each forwarding algorithm is compatible with the SDF forwarding model). This is because the ability to imitate a decision based on a packet header is architecture-neutral. Also, the demultiplexing is based on either the input port or the packet header, and thus demultiplexing is codified in the resulting hardware decision entry. That is, SDF treats the demultiplexing as part of the overall forwarding decision.[6]

To do this in a manner in which each forwarding application gets its fair share of system resources, the system software should segment the TCAM between each of the applications. This requires either the applications be trusted to provide a unique segment ID, or for the system software to be able to determine which application made a particular forwarding decision. Of course this is only a first order protection measure as errant applications could still consume disproportionate amounts of CPU or control bandwidth. Protecting against such cases requires isolation measures, which are orthogonal to SDF and well understood by the research community (see *e.g.*, [24]).

[6]This virtualization approach is similar to FlowVisor: these two pieces of work were done simultaneously. The FlowVisor work explores the applications of virtualization while the study here focuses on the software API and merely identifies virtualization as a natural and straightforward byproduct of software-defined networking.

Figure 2 depicts our virtualization prototype based on *Linux-VServer* [19]. Each forwarding application runs in its own slice, on top of a local copy of system software. These system software instances in the slices are then connected to the master system software instance in the "root" slice, which maintains a table of demultiplexing keys (*e.g.*, VLAN IDs) to uniquely identify the destination slice for incoming packets requiring a forwarding decision. The master system software is also responsible for ensuring fair resource allocation. In our prototype, we assume administrator configures the slices, their demultiplexing key and resource allocation *a priori*. We have used our prototype to run several different forwarding algorithms simultaneously, including XORP, Chord, *i3*, and VL2 [13], and have achieved hardware speeds and good isolation.

Network virtualization doesn't impose hard requirements on the platform virtualization; we only assume that *a)* the master system software runs somewhere, isolated from the forwarding slices, within the physical host, and *b)* there is a communication channel between the master system software and the system software running virtualized forwarding applications. Therefore, depending on the platform virtualization approach, the master system software may run as a Unix process in a special management virtual machine or even within the virtual machine monitor itself. Similarly, the interconnection between the master system software and slices can use the communication mechanism most suitable to the chosen platform virtualization solution.

4.3 Implementing Forwarding Algorithms

In this section, we explore the suitability of our API for implementing various protocols. We describe the implementations of two standard protocols (L2 learning and IP forwarding), as well as three research proposals (SEATTLE, *i3*, Chord). All implementations were built in and tested on our prototype described in Section 7.

4.3.1 L2 Learning Switch

An L2 learning switch operates by maintaining a mapping between MAC addresses and the physical ports on which they can be reached. These mappings are "learned" by watching the source addresses of packets as they traverse the forwarding software. Learned addresses expire after some period of inactivity, and each entry is updated when the switch receives a frame with the same source address through a different port (*e.g.*, during host movement). Incoming packets for which there is no learned MAC address are flooded.

When a packet arrives and there is a mapping for its destination MAC address, an entry is created in the TCAM with the destination MAC address field marked as a dependency. We also mark the incoming port as a dependency to ensure that we can see packets from other hosts on the network. If the switch does not find a mapping, the cached forwarding decision is simply to broadcast the packet. To revoke the entry when the mapping changes, we annotate the decision with a particular state identifier associated to the mapping entry.

4.3.2 IPv4 and IPv6 Forwarding

IP forwarding proved to be a challenging forwarding algorithm to support on our platform. The challenges stem from both the contents of the IP header as well as from the complicated lookup algorithm involved, LPM.

To forward IPv6 packets, the forwarding software has to inspect the destination address and modify the TTL. Thus, the dependencies are the address and TTL, and the rewritten packet header modifies the TTL. IPv4 is more complicated, because in addition the forwarding software must verify the IP header checksum and then recompute the checksum after the TTL has been modified – and therefore, it needs to mark the entire packet header as a dependency (because the entire packet header is involved in the header checksum recomputation). The obvious remedy is to support checksumming and TTL decrementing in hardware, as is currently done. This can be done by providing some basic arithmetic primitives (decrementing, checksum, hash, etc.) that can be applied to specified sets of bits (*i.e.*, the hardware does not have to understand where the TTL field is). The SDF API could be extended with these additional operation-specific calls to convey enough information to the system software from the forwarding application to imitate these operations in hardware, assuming the hardware supports them.

LPM as a lookup algorithm is challenging for our API due to its implicit use of a conflict resolution scheme (longest match) for deciding the correct forwarding action. In our Python implementation of a simplified IPv4 router, the router pre-loads a FIB (from a RIB obtained from RouteViews) and builds a trie. When it is invoked by cache misses, it looks up the trie to figure out the next hop address. Since this next-hop result is valid until the FIB changes again for that entry, the software marks the destination IP address and TTL, and associates the sent packet with the corresponding "IP-to-next hop" mapping entry in the FIB data structure. When updating the FIB, the software creates a new trie and checks whether the new trie yields the same next hop for each IP address in the old trie. If the result differs for an address, the software revokes the corresponding old entry and constructs a new one, otherwise using the old entry in the new trie. To maintain the correctness of the forwarding, the router implementation calls revoke() for all FIB entries sharing the same prefix as the newly added prefix.

In our approach, the software is oblivious to the existence of the TCAM.[7] Further, because the software is deterministically making a forwarding decision over the packet headers, and the system software is revoking these decisions as the system state changes, the TCAM should never contain conflicting entries. The trade-off of using non-conflicting entries is that there is the potential for substantial hardware state explosion which is generally ameliorated through the use of priorities. For example, the default entry in LPM) conflicts with every other entry. In our approach, this would require every non-conflicting destination address prefix which matches the default route to be added as a separate hardware entry.

Although, SDF doesn't assume the use of priorities in hardware classification rules, the system software could maintain the header bit values and the masks of cached decisions in an order suitable for TCAMs to reduce the number of TCAM entries using the following algorithm:

1. First consider only decisions not having a "don't care" bit set in the beginning of the search mask. Assign these decisions to groups based on their longest, first consecutive exact match; the group having the longest exact match of all will have the highest priority and the group with least exact match bits will have the lowest priority.
2. Then assign all the remaining headers (which all have a "don't care" bit or more set in the beginning of their masks) to groups based on the number of consecutive "don't care" bits; the decisions in a group with only one "don't care" bit get a priority number just below the lowest priority assigned for group in the step one. The group with most "don't care" bits gets the lowest priority of all groups processed in the steps 1 and 2.
3. Repeat steps 1 and 2, recursively, for each group to sort the entries within the group. Repeat until no cached decision has search mask bits left.

However, this does not solve the problem of determining how to update TCAM entries without requiring the rewriting of all entries, which could result in relatively long periods of time (even seconds) when TCAMs cannot be used in packet forwarding. We don't present a fast update algorithm here, but view the algorithms for performing incremental updates developed for IP addresses [27] and for general packet classification [28] as excellent starting points.

4.3.3 Floodless in SEATTLE

We also implemented SEATTLE [18] for our prototype. Like *i3*, SEATTLE demonstrates SDF can support hardware forwarding speeds of nonstandard protocols.

SEATTLE performs host discovery without resorting to flooding [18]. It does so by forming a link-layer DHT, that stores information about host locations (*i.e.*, associated switches), IP addresses, and MAC addresses. Briefly, when a switch S_1 detects a directly connected host H_1, it learns H_1's MAC address and IP address from traffic and publishes the $\mathrm{IP}_{H_1} \rightarrow (\mathrm{MAC}_{H_1}, S_1)$ and the $\mathrm{MAC}_{H_1} \rightarrow S_1$ mappings in the DHT. When a remote host H_2 then sends a packet to H_1, H_2's local switch S_2 resolves H_1's address and location through the DHT, and tunnels the packet to S_1 on behalf of H_2. If the mapping changes (*e.g.*, host migration or NIC replacement), S_1 performs DHT operations to update the mapping.

We limit the description of our implementation to location resolution since address resolution is handled in a similar manner in SEATTLE. On receipt of an Ethernet frame from a host, our Python forwarding software first publishes the $\mathrm{MAC}_{\mathrm{SRC}} \rightarrow$ Switch mapping if the address has not been published before. Next, if it finds an entry in its local mapping table, it sends the frame to the cached location (*i.e.*, the associated switch) via the shortest path. To do so, the switch encapsulates the frame by sending it out of a logical port which maintains a tunnel to the destination switch. The software marks the destination MAC of the received packet as a dependency thereby ensuring that the decision remains cached until the mapping changes.

If there is no mapping entry found, the forwarding software initiates the DHT lookup process, and marks the destination address in the packet header as a dependency; then the software sends the packet to a special "null port"; this caches the decision to drop frames destined to the address. This decision is also annotated with state identifiers corresponding to the (pending) mapping lookup sent to the DHT; once the reply arrives (or software timeouts while waiting for it), the cached decision will be revoked. On receipt of a DHT message, our implementation performs the appropriate DHT operation (*i.e.*, insert, delete, lookup). Note that since DHT messages are control messages, they are not subject to caching.

Finally, when the software receives an outer frame, it inspects the encapsulating header to determine whether it requires further forwarding. If the frame requires forwarding, it marks the destination address of the outer header and forwards the packet further without en/decapsulation. Since this decision is valid until the topology changes, the sent frame is annotated by the relevant entries in the data structure holding the information about topology. If the frame does not require forwarding to another switch (*i.e.*, the frame is addressed to the switch itself), the switch marks the destination address of the encapsulating header and sends the frame to a "decapsulation port" (coupled to the actual physical output port connected destined to a host) removing the encapsulation header before sending the packet out on the wire. Forwarding the packet to the decapsulation

[7] Modulo API calls aiding the system in tracking the header bits and internal application state used in forwarding decisions.

port signals the system software to include hardware-implemented decapsulation in the cached decision.

4.3.4 *i3* and Chord

We used our prototype to implement another non-standard design, *i3* [29]. In *i3* servers register listeners—*triggers*—in the network and obtain *trigger identifiers*, which they publish to allow senders to find them. In our implementation, trigger identifiers are published in a Chord DHT [30]. To send a packet to a server registered with *i3*, the sender looks up the server in the DHT by traversing Chord's ring until it finds the server's trigger identifier, which it can use to forward the packet to the trigger. The trigger then forwards the packet to the server.

We've implemented both Chord and *i3* forwarding on SDF. This required matching packet headers below the transport layer at the Chord and *i3* layers, demonstrating the flexibility of the SDF approach. Both Chord and *i3* forwarding require only exact matches on their respective identifier fields. Specifically, Chord's cached decisions include the 8 bit packet type field, the 160 bit Chord identifier, and the 8 bit Chord TTL. For *i3*, the cached decisions include the 32 bit identifier stack size (the number of stacked identifiers) and a 256 bit *i3* identifier. The required TCAM search mask length is therefore well beyond the standard maximum of 576 bits.

5 Performance in Reactive Mode

Our design contains both a software component (the forwarding application and the system software, running on a general CPU) and hardware component (based on a classification engine, in our case a TCAM). If the system is run in proactive mode, and the hardware can store enough state to capture all possible forwarding decisions, then the system will run at hardware forwarding speeds. This is how we imagine the system will be run in performance-critical scenarios (such as the Internet backbone). If the system is run in reactive mode, as would be appropriate for most experimental uses where performance is not critical but a simple and convenient programmatic API is essential, then the resulting speed depends on how many packets are handled by software, and the relative speed of the hardware and software forwarding actions. This is our focus in this section.

While the ability of commodity CPUs to perform packet forwarding has increased significantly over the last few years, TCAMs remain significantly faster. For example, an 8-core PC supports forwarding capacities of 4.9 million packets/s [12], while modern TCAMs can achieve rates of 150–200 million packets/s.[8] Even special purpose general processors which include network hardware on die (*e.g.* [9]) are an order of magnitude slower than TCAMs.

In order to marry a slower software path with a much faster hardware path in a manner that takes full advantage of the system, the ratio of the number of packets processed by hardware versus software must be commensurate with the differential between the speeds of the two layers.[9]

For purposes of analyzing these issues we assume software forwarding performs two orders of magnitude slower than the forwarding engine implemented in the hardware. This assumes that the switch employs one or more high-speed multicore CPUs (as is true for many newer switches [2]), and is a conservative estimate of the speed ratio given the TCAM and CPU figures cited above. With this assumption, in order to fully realize the forwarding potential of our system, the cache hit rates must exceed 99%. The hit rates depend on the nature of the traffic, and the size of the cache.

We now analyze hit rates using standard L2 and L3 forwarding algorithms on real network traces to see whether a 99% hit rate is a reasonable expectation for feasible cache sizes. We used the set of traces described in Table 1, and we assumed 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.[10] We warmed up the cache for half of the trace, and then measured the hit-rate for the remainder of the trace.

L2 learning. Our MAC learning implementation uses 15 second timeouts. For analysis, we used the first trace in Table 1, and the resulting cache hit rates are shown in the left-hand graph of Figure 3. The cache miss rate is less than 1% for cache sizes larger than 64, and is nearly 0.01% for a cache with 256 entries. Note that these TCAM state sizes are significantly smaller than today's commercial Ethernet switches, which commonly hold 1 million Ethernet addresses (6MB of TCAM). Thus, SDF in reactive mode would easily reach the requisite 99% hit rate with very moderate state requirements in this particular enterprise setting. We have looked at other enterprise traces, and found similarly encouraging results.

IPv4 forwarding. We next explore the cache behavior of standard IPv4 forwarding, assuming that the hardware supports TTL decrementing and checksum recomputation (so the forwarding decision depends only on the destination address). We only had one trace with an associated FIB, which is the ISP trace in Table 1. In order to obtain

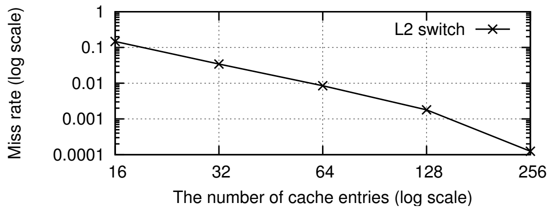
[8]The classification rate largely defines the maximum hardware forwarding speed since applying packet modifications is easier than classification. Similarly, the high-speed switch fabric connecting ports is not a limiting factor.

[9]In addition, the bus between the hardware and software must not be saturated, and the hardware should be able to update the classification rules as fast as the system software updates them, and these too depend on the cache hit rate.

[10]Routes to popular destinations are quite stable [25].

	L2 trace	L3 traces						
Name	*Enterprise*	*OC12* [4]	*OC48* [5]	*OC192-A* [6]	*OC192-B* [7]	*OC192-C* [7]	*ISP*	*ISP2*
Source	LBNL	CAIDA AMPTH	CAIDA	Equinix Chicago	Equinix Chicago	Equinix San Jose	ISP Europe	ISP USA
Date	Dec 2004	Jan 2007	Aug 2002	May 2008	Feb 2009	Mar 2009	Mar 2005	Jan 2010
Duration	1 hour	2 hours	1 hour	1 hour	1 hour	1 hour	NA	1 hour
# packets	1,977,405	29,092,430	419,720,983	741,205,934	111,839,231	1,551,424,452	17,526,284	598,534,072
Anonymized	Yes	Yes	Yes	Yes	Yes	Yes	No	No

Table 1: Traces used in the cache hit-rate evaluations. All the traces except the last contained full packet headers and timestamps. The last trace only included a series of destination addresses without timestamps. However, because this last trace was not anonymized (but the last few bits were elided), we could use the accompanying FIB to make accurate forwarding decisions; for the other traces we had to assume that forwarding decisions were done at either the /16 or /24 granularity.



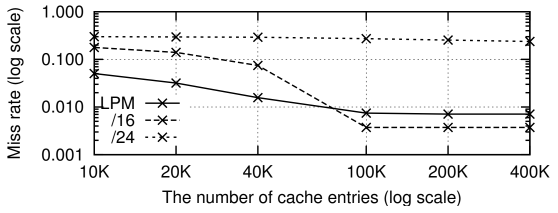


Figure 3: Simulations of cache miss rates for (left) L2 learning with the enterprise trace and (right) LPM using the ISP trace.

conservative results, we used no warm-up period, so every cache entry experiences at least one miss. The cache hit rates of this analysis are shown in the right-hand graph of Figure 3. There are three curves: one assuming that the forwarding decisions are based on the FIB, one assuming that the forwarding decisions are based on /16 prefixes, and one assuming that the forwarding decisions are based on /24 prefixes. For now we focus on the FIB-based curve, since that represents what would happen in practice. Here, the miss rate is less than 1% when the cache holds more than about 64k entries, which is a small amount of state for an ISP-class router. We do not know the link's speed for this trace, so to investigate hit-rates on very high-speed links we used several traces from CAIDA. For these traces we did not have an associated FIB (because the traces were anonymized); we therefore measured the hit-rates assuming that the forwarding decisions were done on either a /16 or /24 granularity. The data from the ISP trace indicates that, in that example, the /16 granularity is a better estimate of the hit rate, and that the /24 granularity is very conservative (roughly two orders of magnitude higher miss rate than the FIB results for large caches).

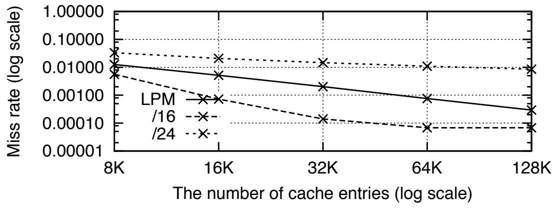


Figure 4: LPM using the ISP2 trace.

To support our assumption, we repeated the same experiment on the *ISP2* trace. The trace was collected by a large domestic ISP and aggregates 10-30K downstream customers. The trace contains IP addresses that are not anonymized and has a matching FIB.[11] Figure 4 presents that the result is consistent with the previous measurement.

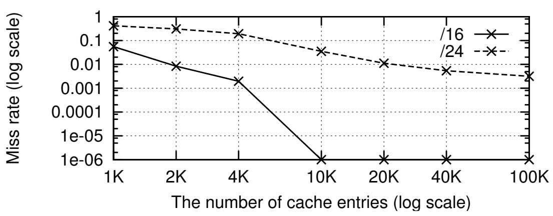
Figure 5 shows the results for the OC-48 trace and the OC-192C trace (the results for the other OC-192 traces are qualitatively similar). When using /16 forwarding, even tiny cache sizes (2k entries for OC-48 and 1k entries for OC-192; the 1k point lies off the graph in Figure 5) are sufficient to achieve 1% miss rates. For /24 forwarding (which the ISP trace suggests is a very pessimistic estimate), cache sizes of 20k entries and 40k entries are required for 1% miss rates, which is again quite reasonable for ISP-class routers.

If these traces are representative, these results suggest only moderate-sized caches can achieve sufficiently high cache hit rates in both enterprise and ISP networks.

Performance under stressful conditions. We now discuss two situations where caching's performance may get stressed: when the software FIB changes and when the switch is under attack. First, when the FIB changes, this will change a large number of forwarding entries. In traditional flow caching, this requires flushing the cache and enduring a high miss rate until the cache is warmed up again. However, in our approach, when the routing state changes, the application can push updated decisions to the hardware after revoking the invalid decisions, rather than merely revoking them. This does require a fast channel to update the hardware component, which may be the limiting factor, but there is no need for a long cache warmup period after a routing event.

Second, in a denial-of-service attack, attackers can inject spurious traffic in the hope of dislodging valid cache entries, which would increase the miss rate. Once the miss rate goes over 1% (assuming the two order of

[11]The ISP generously has run our simulator on the trace and we have been reported only the cache hit rate result.



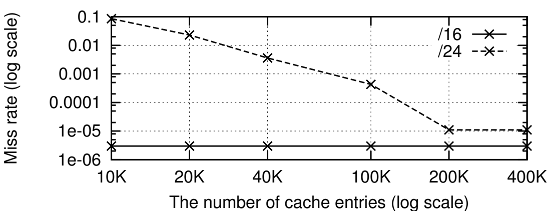
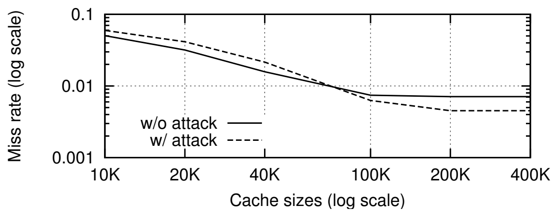


Figure 5: Cache miss rates with forwarding on the /16 and /24 granularities for the OC-48 (left) and the OC192-C trace (right).



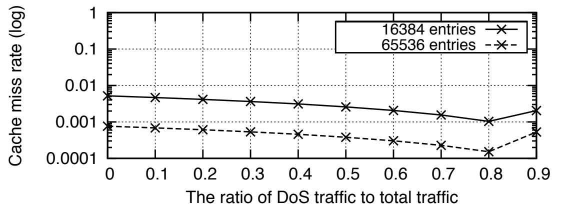


Figure 6: Cache miss rates with (and without) random DoS traffic comprising 50% of the traffic for the ISP trace (left) and cache missrates with various random DoS traffic rate for the ISP2 trace(right).

magnitude difference between line speeds and the CPU), then some of the cache-miss packets will be dropped. We now investigate how effective such an attack might be, assuming the cache uses a LRU replacement policy.[12]

Using the ISP trace in Table 1 (again, with no warm-up period), we examine the effect of a DoS attack that sends traffic with random destinations. The left graph in Figure 6 compares the miss rate with no attack to the miss rate when the attack traffic comprises half of the total traffic. With an attack of this magnitude, overloading the link is likely to be more of a problem of cache misses, but we wanted to look at an extreme scenario. Somewhat to our surprise, this massive attack only has a very limited impact on the miss rate; this is because the long-tailed distribution of network destinations makes it unlikely that any of the frequently used cache entries is ever evicted. In fact, for the larger cache sizes, the miss rate with the attack traffic is *lower* than without the attack traffic (this is because the random addresses tend to fall on the large prefixes, whereas the trace traffic more systematically explores the address space). The right graph in Figure 6 also conforms to the assumption and indicates that random DoS attack could hardly impose an impact on the cache performance.

We don't claim to have solved all performance problems associated with caching. However, if caching is mainly used in environments where performance is not critical, then these issues discussed above (and other performance issues) may not be fatal. While the proactive mode is clearly safer, the reactive mode may still be an attractive choice considering how small the state requirements are to achieve low miss rates.

[12]The network can take more explicit measures against such attacks, such as rate-limiting the number of new flow entries from any port, which would limit the scope of such an attack. However here we just focus on the attack traffic's impact on the cache miss rate.

6 System Design

In this section we first describe the API we designed while implementing our system. This API reflects lessons learned while implementing a range of protocols, but it is only one example of many possible software interfaces. We then discuss the system software design, followed by an overview of the hardware design.

6.1 API

Basic operations. For basic packet sending and receiving, our API is modeled around a standard datagram socket interface. To gain access to traffic, a forwarding application invokes listen() while providing a callback to be signaled on each new received packet for which there is no matching hardware entry and which is ready for the application to retrieve using receive(). On packet receipt, the forwarding application can then process the packet internally (*e.g.*, if the packet is control traffic), or it can make a forwarding decision, modify the packet, and send it out of one or more ports through a call to send().

In addition to basic sending and receiving, the API provides methods for the forwarding software to declare the state dependencies of the forwarding decision. This includes methods for annotating which header bits and what application state the decision relied on, and signaling when the application state has changed (to induce a revocation of any cached state). Modifications to packets are mimicked by the system software by simply copying the modified outgoing header bits of each sent packet and forcing an overwrite of them for every match. Because these additional methods comprise the portion of the API

	Function signature	*Description*
Core	listen(callback)	Register a callback to be signaled on every packet ready to be received.
	read()	Receive a packet; return values are *(in_port, in_pkt)*.
	send(out_pkt, final)	Send a packet. If final is true, the system knows the decision is complete.
InPkt	mark(in_pkt, bit_pos, len)	Mark packet header bits as used in a forwarding decision.
	mark_ingress(in_pkt)	Mark the ingress port as used in a forwarding decision.
	unmark(in_pkt, bit_pos, len)	Unmark matching bit markings, if any.
	clear(in_pkt)	Cancel all earlier bit markings.
	new_in_pkt(in_port)	Prepare a new synthetic inbound packet. Used in proactive decisions.
OutPkt	new_out_pkt(in_pkt)	Prepare a new outbound packet. If it is to be forwarded and not self-generated a reference to a received packet is required to copy the packet.
	register(out_pkt, state_id)	Register application state used in a decision resulting in this packet.
	add_destination(out_pkt, port)	Add a destination port to send the packet to.
	clear(out_pkt)	Cancel all earlier state bindings, bit modifications, and port additions.
Control	new_state_id()	Construct a state identifier to store into application state data structures.
	removed(state_id)	Inform system about changed application state.
	get_ports()	Obtain a list of identifiers of ports the system has.
	add_port(type), del_port(id)	Create/remove a virtual port. Type defines the type of tunneling/encryption (possibly implemented in hardware) to use.
	set_properties(id, key, val)	Set port properties. For example, a port representing an IPsec tunnel requires configuring encryption, as well as a tunnel end-point.
	get_properties(id, key)	Get a current port property value.
	get_stats(port_id)	Get the latest port statistics. Format of statistics is port type specific.

Table 2: Forwarding API. The *Core* calls are to receive/send packets, *InPkt*/*OutPkt* are for packet operations. *Control* calls are for control plane.

which constrains the programmer beyond the traditional software model, we describe them in more detail below. For a complete summary of the API, see Table 2.

Annotating dependencies. It is difficult to build a system software layer that can automatically – without any help from the developer – infer header and state dependencies without incurring exorbitant runtime overheads. Rather than aiming at full transparency, we chose instead to put the onus on the programmer to explicitly mark headers and state used in forwarding decisions.

This is done through so-called *state identifiers*. State identifiers are used to track application state that is used to make forwarding decisions. The canonical example of such state is the routing table. For each uniquely identifiable component of system state used for forwarding decisions, such as a RIB entry, the forwarding application must request and associate a new state identifier from the API.

This identifier is used to signal state changes to the system. For example, when removing an entry from an internal data structure representing the forwarding table, the application is responsible for calling the removed() call on the associated state identifier. This will trigger the system to revoke all hardware entries associated with that particular entry. If instead of deleting the entry, the application modifies it, the application still must obtain a new identifier to replace the old one which is now invalid.

While making a forwarding decision for a received packet, the application is required to register() any state it uses in a decision; since the state identifiers are conveniently stored in the data structures, the application can register() the state used into the outbound packet (abstraction) while traversing the application data structures and making the forwarding decision. Once the application then send()'s the packet, the system can record the state identifiers related to the decision to be cached. If the application subsequently removes the data structure entry (and informs the API about it), the system can identify the cached decisions to remove from hardware, and hence, maintain the correctness of the forwarding function.

Similarly to annotating state, in order to provide bit-level dependency information for the packet header, the application uses the mark() method to mark the header bits it used in making a forwarding decision on a given packet. For headers in which the forwarding application marked the bits, the system software will generate a TCAM entry whose values matched those of of the packet. All other header bits are set to "don't care".

The system software can infer packet header modifications by observing the packets as they are sent by the application. This works so long as for each incoming packet, the resulting outgoing packets are modifications of the original packet. Unfortunately, this is not always the case. In certain protocols, incoming packets require intermediate packets to be sent before forwarding decisions are made. For example, if the MAC address is not cached for a given next hop IP address, the packet is queued and an ARP packet is sent instead. To aid in such scenarios, our API considers the special of case of no packet header bits nor state dependencies marked as a sign not to cache anything. Hence, the application can respond to the ARP request by itself.

Finally, the forwarding application must send the packet to one or more ports. As with standard forwarding software, ports are represented via a programmatic abstraction. When a forwarding application sends a packet out of a port, the system software uses the information annotated by the application during the forwarding decision and sets up a hardware entry correspondingly.

Prioritizing control protocols. Networking forwarding software can often be divided into a control plane and a data plane. Under this model, the data plane is responsible for performing packet lookups against one or more data structures which are maintained by the control plane. The control plane maintains the data structure(s) by sending control traffic between participating nodes.

In our approach both standard network traffic and control traffic share the (limited) bandwidth between the hardware and the CPU. Further, on failure conditions (*e.g.*, link failure), many hardware entries may be invalidated causing a flood of packets to the CPU. This has the potential of starving out the control traffic when it is most needed. To prevent such cases, the API provides a method for setting up entries in hardware that have prioritized service to the CPU. Thus, during failure conditions such as routing instabilities, the control traffic is given precedence allowing the system to converge quickly.

The entries for control traffic have the same expressibility as other hardware entries (they can be defined over any header field or incoming port). However, instead of sending to an egress port, the forwarding application can send the control traffic packets to a logical "control" port, which represents ingress queues given weighted priority compared to the queue reserved for sending data plane packets from hardware to CPU.

The method for identifying control traffic is identical to other forwarding operations. If the data plane makes a decision to send a packet to the logical "control" port, a corresponding entry to the hardware is put in place by the system software. Generally these entries are maintained in hardware for the lifetime of the forwarding application.

Proactive decisions. In the reactive mode, the API operates in pull mode: whenever the system software receives a packet from the hardware requiring a forwarding decision, it asks the application for the decision. While this allows the application to remain oblivious to the decision replacement process, the simplicity comes with a cost: a new decision about a a revoked decision can only be made when a new packet requiring the old entry arrives. This causes some additional forwarding delay for this first packet, as it takes some time for the decision to be made and the entry installed, but for some applications this is of little concern. However, if the stream of packets that would use this revoked entry exceeds the capacity of the bus and/or the CPU, the system has to drop packets until the new entry has been inserted into the TCAM.

For these demanding environments and applications, the API provides a mechanism to proactively push replacement decision(s) at the same time the corresponding TCAM entries are removed. This is done by having the application create an appropriate synthetic packet (or packets, as necessary) whenever an entry is revoked, so a new entry is installed at the same time the old one is revoked. The system software does not forward these synthetic packets, it only uses them to construct and enqueue update(s) to the TCAM.

6.2 System Software

The system software is responsible for implementing and exposing the programmatic API used by the forwarding application, and directly managing the hardware. While designing the API, we tried to strike a balance between relatively unconstrained program design, and the ability to efficiently implement the system software. The API itself imposes few restrictions on how the system software realizes the required functionality for applications.

Language support. One can consider the proposed API as the low-level "Sockets API" for routers/switches; in a manner similar to the Sockets API, how the functionality is exposed for applications depends on the programming language and development environment in question. Moreover, how the API relates to other interfaces provided for the forwarding applications also depends on the chosen language and environment. For example, for C/C++ applications, it is natural to align our API next to the Sockets API and provide it as a part of the basic system libraries.

Application portability. From an application perspective, the system software implementation is irrelevant as long as the application's language is supported and the different API implementations for the same language remain binary compatible. In Section 7, we discuss our implementation in which the system software is implemented entirely in user-space providing both a C and Python interface, both backed by an FPGA-based hardware forwarding plane.

6.3 Hardware

At a component-level, the idealized hardware design is the standard high-speed, high-fanout distributed forwarding architecture (as shown in Figure 7). A system-level CPU manages the shared state between CPUs local to each line card. The line card-local CPUs directly manage a packet classification engine, which is used by the high-speed packet processing logic to perform the lookups. The line

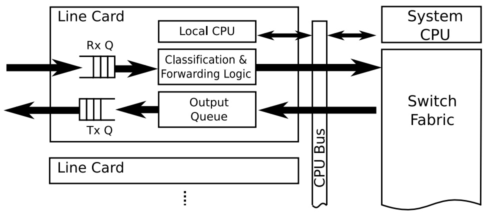


Figure 7: A distributed forwarding architecture.

cards also contain high-speed random access memory to store the forwarding actions; the packet processing logic uses the random access memory to retrieve the required modifications for packets, as well as to determine the destination port(s) to forward the packet to. All line cards are interconnected via a high-speed backplane.

A less scalable version can be implemented in a manner similar to commodity switches and routers available today. In such an instantiation, a control CPU controls a single hardware forwarding engine (consisting of classification engine and high-speed packet processing logic). This is the approach we took with our prototype implementation which we describe in Section 7.

In these designs, the feasibility and implications of performing packet classifications using wide search masks are most essential. While our design does not dictate a particular packet classification component, for the following discussion we consider the use of TCAMs since they are widely used in networking hardware.[13]

The largest TCAMs on the market today are 36Mbits in size and have an entry with of 288 or 576 bits. Assuming standard protocols, a 576 bit entry is sufficient for lookups covering the Ethernet header, the IP header (without options), as well as the TCP/UDP headers, while 288 bits is sufficient for both the Ethernet and IP headers. Given these wide entries, the maximum number of rules a TCAM chip can store is either 128k or 64k, respectively for 288 and 576 bit entries. In addition, TCAMs are designed to support stacked use in backs of 4 or 8, allowing for the (expensive) possibility of building a forwarding engine which contains 512K, 576-bit entries.

While TCAMs can't match header values against 576 bit wide entries in a single clock cycle, they still manage extremely high throughput. For example, to match 576 bits, a modern TCAM typically requires 4 clock cycles. Therefore, at 200MHz a chip which sells for roughly $300 can support 40Gbps of bandwidth. A TCAM with 288 bit wide entries uses less for lookups, and therefore, can sustain 80Gbps. Given our design, if the additional hardware logic does not introduce additional overhead, a wide TCAM (576 bits) can support four 10Gbps ports, or a more standard configuration of 24, 1Gbps ports.

The main complexity of the system software is the optimal management of entries in TCAMs and forwarding actions in SRAMs on line cards; *i.e.*, keeping the cards busy forwarding without dedicating extensive periods of time to updating the entries in TCAMs and in SRAMs. As long as the TCAM entries are non-conflicting, the system software can implement a simple cache replacement policy and remove the least used entries as necessary; this is possible since as the order of non-conflicting TCAM entries doesn't matter. If the number of required TCAM entries is less than the number of TCAM entries available in total, the task is even simpler. The maximum update speed is determined by bandwidth available on CPU bus towards line card(s) (and by the CPU itself).

A commonly raised concern with the use of TCAMs (besides price) is power consumption. It is true TCAMs consume more power and are more expensive than either DRAM or SRAM, however a single TCAM can sustain considerable traffic. When compared to a commodity processor (which has been argued as a flexible alternative to standard switching hardware [11, 12]), TCAM power use is far more modest. For instance, in practice a TCAM can consume up to 30 watts, which is significantly less than the typical power consumption of a Pentium 4 at 2.8GHz rated at 68 watts. Certainly when measured in terms of power consumed per packet forwarded, TCAMs are far more attractive than commodity processors.

7 Implementation

We implemented a prototype system to validate the design, as well as to better understand the implications of our design decisions. We implemented the API with two language bindings, C and Python. We choose to support the latter because we believe it emphasizes the flexibility of the approach. In particular, it demonstrates that we can use a (very) high-level language for implementing forwarding algorithms and still achieve wire forwarding speeds. Indeed, even though Python is a scripting language and performs roughly 20 times slower than an equivalent C program, we were able to get hit rates which allowed us to take full advantage of the hardware datapath.

The Python forwarding applications run on a Python interpreter embedded in a C system software implementation. The system software manages two "line cards" we implemented (for now, we don't support their simultaneous use): one running as a process in user-space and written in C (and running in the same host as the system software, connected over TCP); and one implemented as a NetFPGA device. The user-space line card uses the host network interfaces; its implementation is straightforward and not of independent interest. We describe the NetFPGA implementation below.

NetFPGA prototype. We implemented a NetFPGA based hardware "line card" with a simulated TCAM with 32 x 512 bit wide matching entries and four 1Gbps

[13]Some of the information here are gathered from private discussions with a TCAM vendor. Public information regarding commercially available TCAMs are available at [15, 23].

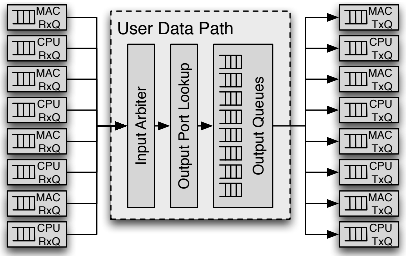


Figure 8: Packet processing pipeline of the NetFPGA prototype. User data path refers to custom logic on the FPGA.

Ethernet ports. Figure 8 depicts the packet processing pipeline implemented in our NetFPGA prototype. The pipeline is organized in five stages. The first stage, the *Rx Queues*, reads packets from the CPU or from the Ethernet. The *Input Arbiter* stage selects which of the *Rx Queues* to service, and pushes the packet to the *Output Port Lookup* module. The Output Port Lookup module looks for matches and does any replacements required on the packet. The packet is then stored in the *Output Queues* module until the output port is ready to receive it. The *Tx Queues* are responsible for delivering the packet from the *Output Queues* to the CPU or to the Ethernet.

As a packet arrives into the Output Port Lookup module it gets stored in an input FIFO and at the same time is sent to a "rule selector" that creates 64-bit words to match against. The module then reads the rule from the memory and compares it to the output of the rule selector using the mask read from the memory. Each packet match requires nine 64-bit rule words to be matched. The first word contains the input port and the other eight are packet data words. To simplify the matching, the memory is organized in 512 byte blocks. Entries are written vertically into memory so that the first memory block contains the first rule word (64 bits) of both match data and match mask, of all 32 entries. Similarly, the subsequent memory blocks contain the remaining eight rule words. Therefore, as the words read from the packet (header) arrive from the rule selector, they can be matched a block-by-block to all rule words. However, we note this simple implementation doesn't scale to larger entry numbers well; as the number of entries grows, the number of required clock cycles to match a packet increases accordingly.

The memory is also used to store the lookup results: the replacement data, the replacement mask, and the hit packet and byte counters. The entries are written horizontally such that each memory word contains all the data for a single match. When a match is found, the Output Port Lookup module reads the packet out of the input FIFO, and uses the result from the match to replace the first 64 bytes of the packet headers as required. If a match is not found, the module reads the packet from the input FIFO and sends it to the CPU over PCI.

Microbenchmarks. The table below contains latency and throughput rates for cache entry insertion and deletion for our NetFPGA prototype. The entry insertion is far slower because it requires writing the full lookup entry, the header modifications, and it requires overwriting the packet and byte counts. Deletion on the other hand simply overwrites the enable bit for the entry.

	Overhead	*Latency*	*Throughput*
Entry insert	300 bytes	55.5 ms	18,018/s
Entry delete	4 bytes	740 ns	1,351,351/s

We also performed throughput testing for the data path using a hardware packet generator which can perform line speed testing. Tests were performed with full line rate on all four ports. Results include the Ethernet CRC, the inter-packet gap, and the packet preamble.

Packet size (bytes)	64	128	512	1500
Throughput	3.8G	3.8G	3.8G	3.8G

8 Concluding Comments

The SDF approach we have presented here contains nothing new; it merely glues together two well-known components (software decisions and hardware classification) into a complete forwarding solution. We have designed a high-level API that allows forwarding solutions to be developed in a high-level language independent of the low-level hardware implementation. The approach can be implemented on top of OpenFlow or other similar hardware interfaces, or could be ported directly to commercial switch SDKs.

Our preliminary work suggests that this combination of known components provides a flexible packet forwarding platform that, because it is based on standard hardware (*e.g.*, TCAMs), should offer competitive cost/performance. We think that SDF may prove quite useful in building experimental networks. Its relevance to production networks is more questionable, as the need for flexibility in that context remains an open question.

References

[1] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. *Computer*, 38(4):34–41, 2005.

[2] Arista Networks. `http://www.aristanetworks.com`.

[3] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM*, 2006.

[4] `http://www.caida.org/data/passive/passive_2007_dataset.xml`.

[5] `http://www.caida.org/data/passive/passive_oc48_dataset.xml`.

[6] `http://www.caida.org/data/passive/passive_2008_dataset.xml`.

[7] `http:// www.caida.org/data/passive/passive_2009_dataset.xml`.

[8] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *HotNets*, Oct. 2008.

[9] Cavium Networks. `http://www.caviumnetworks.com/`.

[10] ControlPoint Developers Alliance. `http://www.fulcrummicro.com/cda/`.

[11] M. Dobrescu et al. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP*, 2009.

[12] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards High Performance Virtual Routers on Commodity Hardware. In *CoNEXT*, 2008.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[15] Integrated Device Technology (IDT). `http://idt.com/`.

[16] J. Kelly, W. Araujo, and K. Banerjee. Rapid Service Creation using the JUNOS SDK. In *PRESTO*, 2009.

[17] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *PAM*, 2009.

[18] C. Kim, M. Caesar, and J. Rexford. Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises. In *SIGCOMM*, 2008.

[19] Linux-VServer. `http://www.linux-vserver.org/`.

[20] G. Lu, Y. Shi, C. Guo, and Y. Zhang. CAFE: A Configurable pAcket Forwarding Engine for Data Center Networks. In *PRESTO*, 2009.

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

[22] J. C. Mogul et al. Orphal: API Design Challenges for Open Router Platforms on Proprietary Hardware. In *HotNets*, 2008.

[23] Netlogic Microsystems. `http://netlogicmicro.com/`.

[24] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *OSDI*, 2006.

[25] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP Routing Stability of Popular Destinations. In *IMW*, 2002.

[26] G. Schaffrath et al. Network Virtualization Architecture: Proposal and Initial Prototype. In *VISA*, 2009.

[27] D. Shah and P. Gupta. Fast Incremental Updates on Ternary-CAMs for Routing Lookups and Packet Classification. In *HotI*, 2000.

[28] H. Song and J. S. Turner. Fast Filter Updates in TCAMs for Packet Classification. In *GLOBECOM*, 2006.

[29] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *SIGCOMM*, 2002.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.