Yale University
Department of Computer Science

Scaling Software-Defined Network Controllers on Multicore Servers

Andreas Voellmy[1] Bryan Ford[1] Paul Hudak[1]
Y. Richard Yang[1]

YALEU/DCS/TR-1468
July 2012

This work was supported in part by gifts from Microsoft Research and Futurewei and by NSF grant CNS-1017206.

[1]Department of Computer Science, Yale University, New Haven, CT.

Scaling Software-Defined Network Controllers on Multicore Servers

Andreas Voellmy* Bryan Ford* Paul Hudak* Y. Richard Yang*

July 2012

Abstract

Software defined networks (SDN) introduce centralized controllers to drastically increase network programmability. The simplicity of a logical centralized controller, however, can come at the cost of controller programming complexity and scalability. In this paper, we present McNettle, an extensible SDN controller system whose control event processing throughput scales with the number of system CPU cores and which supports control algorithms requiring globally visible state changes occurring at flow arrival rates. Programmers extend McNettle by writing event handlers and background programs in a high-level functional programming language extended with shared state and memory transactions. We implement our framework in Haskell and leverage the multicore facilities of the Glasgow Haskell Compiler (GHC) and runtime system. Our implementation schedules event handlers, allocates memory, optimizes message parsing and serialization, and buffers system calls in order to optimize cache usage, OS processing, and runtime system overhead. We identify and fix bottlenecks in the GHC runtime system and IO manager. Our experiments show that McNettle can serve up to 5000 switches using a single controller with 46 cores, achieving throughput of over 14 million flows per second, near-linear scaling up to 46 cores, and latency under 10 ms with loads consisting of up to 1500 switches.

1 Introduction

Network systems are becoming more feature-rich and complex, and system designers often need to modify network software in order to achieve their requirements. Software-defined networking attempts to move as much network functionality as possible into user-definable software, making more of the network system components programmable. In particular, SDN architectures introduce a centralized control server (controller) to allow potentially dramatically simplified, flexible network programming.

Unfortunately, as the network scales up—both in the number of switches and the number of end hosts—the SDN controller can become a key bottleneck. Specifically, Tavakoli et al. [23] estimate that a large data center consisting of 2 million virtual machines may generate 20 million flows per second. On the other hand, currently controllers, such as NOX [13] or Nettle [25], are able to process on the order of only 10^5 flows per second [23].

In this paper, we address the preceding issue by designing and implementing an SDN controller framework that is highly scalable and provides a relatively simple and natural programming model for controller developers.

This work was supported in part by gifts from Microsoft Research and Futurewei and by NSF grant CNS-1017206.

*Department of Computer Science, Yale University, New Haven, CT.

The starting point of our design is two observations on the current Internet architecture. First, the current Internet architecture is highly scalable for key network functions. For example, in the link layer, the key host-location ARP (address resolution protocol) table management function is distributed at individual network forwarding elements (switches) and hence can naturally scale as a network grows larger; in the network layer, the key function of computing forwarding information bases is also naturally distributed at individual routers, where each router uses Dijkstra to compute its own shortest path tree. Second, a major source of difficulty of the current network architecture, however, is the distributed state of its control plane. For example, the partial views of ARP tables at individual switches may lead to unnecessary network flooding; the consistency issues of distributed topologies and route computation at individual routers are a major limit on existing network control plane design.

Based on the observations, our strategy, therefore, is to preserve the scalability structure in today's network architecture at a controller, while hiding its distributed nature. Specifically, we design a controller framework that allocates independent and concurrently operating computation resources to each network forwarding element; at the same time, we introduce a global shared memory to simplify sharing of global network state. We implement this strategy using shared memory multicore servers, which provide abundant computational resources (currently available systems provide upwards of 80 processor cores) and low latency, high-bandwidth shared global memory. In a nutshell, our approach involves virtualizing the distributed control logic in today's architectures into a multicore server, and replacing distributed message-passing algorithms with shared memory access to provide consistent sharing of global network state.

Moreoever, having done this, network algorithm developers can easily introduce new computational elements which do not directly correspond to components in today's architectures. For example, a network security component might run concurrently with switch control logic to detect a variety of security conditions, which it can then easily signal to switch controllers using shared memory data structures, rather than complex distributed protocols.

We realize this vision with a programming framework, McNettle, implemented in Haskell. This framework provides programmers with a simple language for writing event handlers for switch-initiated interactions and facilities for programming background processing that can issue commands for switches. This organization allows switch event handlers to access local private state without synchronization. Event handlers may make use of shared state as well, typically using some synchronization methods to ensure correct manipulation. As mentioned above, we emphasize STM for this, but do not prohibit the use of lower cost methods where appropriate, such as locks or compare-and-swap operations.

McNettle efficiently executes user-specified event handlers on a multicore server, handling issues of scheduling event handlers on cores, managing message buffers, and parsing and serializing control messages in order to reduce memory traffic, core synchronizations, and synchronizations in the Haskell runtime system. As a result, users can write network control algorithms in Haskell using McNettle that can handle loads expected in data centers with thousands of switches and hundreds of thousands of virtual machines and which run on currently available multicore servers. We demonstrate our system by presenting several substantial controllers in McNettle, in each case showing how the McNettle API and the concurrency management tools of Haskell provide convenient methods of writing correct concurrent controllers. We demonstrate throughput scaling through 46 cores to more than 14 million flows per second, several times as much as the current state-of-the-art shared memory multithreaded OpenFlow controller. We also demonstrate latency under 0.5 ms for loads with hundreds of switches and latency under 10ms for loads with thousands of switches.

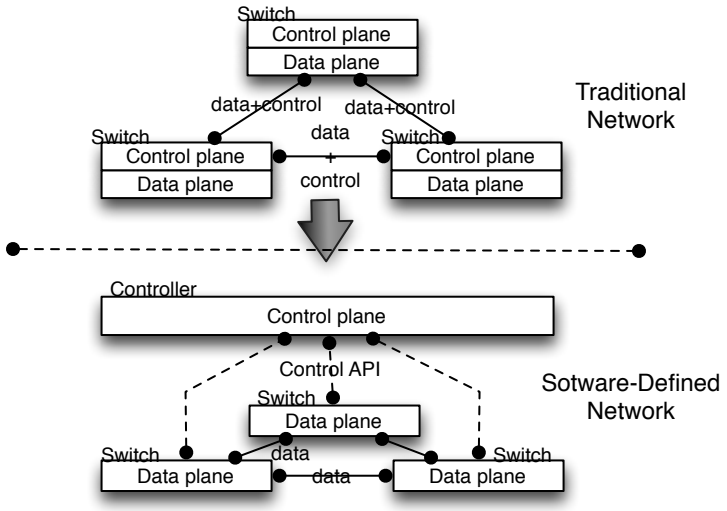

Figure 1: The top diagram illustrates today's network architecture with distributed switches that combine forwarding and network control functionality. The bottom diagram illustrates the Openflow architecture, where the control functionality is moved to a remote control server.

2 SDN & OpenFlow

A SDN consists primarily of forwarding elements, or switches, and a control system, or controller, and an interface between them, and is illustrated in Figure 1. The OpenFlow [22] protocol defines an interface between switches and controllers and we use this protocol throughout the paper. With OpenFlow, switches establish essentially permanently open TCP sessions with controllers over a control network (typically a designated VLAN). Controllers send commands over these sessions to modify switch *flow tables*, which consists of a sequence of conditional forwarding rules, where the condition is expressed as tests against packet header information. OpenFlow switches notify controllers of key events, the most important and frequent of which is the packet-miss event, which occurs when a packet arrives at a switch with no applicable forwarding rule. When this event occurs, the switch sends a message to the controller with some metadata about the event and a prefix of the packet. A controller can then respond with a forwarding command for the packet and optionally a new conditional rule to install. In addition, a controller may issue table modifications to several switches in response to a packet-miss event in order to provision a path through the network for the flow.

3 McNettle Design

McNettle is an *extensible* and *scalable* SDN control system built on OpenFlow switches that takes advantage of multicore servers to implement high throughput and low latency control servers with global state visibility. The high-level design of McNettle relies on two observations; first, that multicore architectures provide sufficient computational, memory, and IO resources to handle demanding network control applications. Second, these architectures allow us to provide a convenient programming environment consisting of low latency shared state and general-purpose computational capabilities. table To see that multicore servers are capable of handling demanding SDN workloads, we consider the data center workload described in [23], consisting of 20 million flow setup requests per second. In OpenFlow, these requests and response require

Application	Characteristics	McNettle Support
Layer 2 Learning	High-rate table updates and lookups	Low latency global state with low overhead synchronization
Routing	Algorithms, parallel computation	Functional programming, parallel evaluation, synchronization
Bandwidth Reservation	Complex global data update	Software transactional memory
Security	External triggers	Background threads with access to shared state & switch commands
Flow Path Provisioning	Communicate with many switches	Efficient support for race-free communication with switches by concurrent threads.

Figure 2: Example SDN controllers, their key features, and the McNettle features supporting these applications.

messages whose size totals approximately 100 bytes. Therefore, the controller may need to process control streams in aggregate of 2 GB/s. Currently available multicore servers, in particular the server we use for our evaluation and which we describe in Section 6.1, can be equipped with multiple 10Gbps network interfaces connected to CPUs and memory with PCIe connections providing tens of Gbps bandwidth, providing sufficient IO bandwidth for these applications. Furthermore, nominal and measured memory bandwidths in our server exceeds 30 GB/s (measured using the NUMA-STREAM [6] tool), providing enough memory bandwidth to transfer control messages into and out of main server memories. We therefore expect that control programs on these architectures can be made to be CPU-bounded, which is ideal, since multicore architectures are expected to continue to increase core counts, and hence aggregate computational capacity over the near future. Therefore, provided the control system uses the memory system efficiently to avoid memory bottlenecks and has sufficient parallelism to take advantage of the aggregate computational capacity of multicore servers, it will be capable of handling the target workload.

McNettle meets these challenges by exploiting switch-level parallelism, processing streams of messages from different switches concurrently. This design allows McNettle to process each control message on a single core, avoiding transferring control messages from one core to another and incurring extra memory transfers and chip interconnect bandwidth. It provides abundant and fine-grained parallelism since SDN networks will consist of many more switches than cores and because an individual switch will generate traffic at a rate that can easily be processed by a single processor core; Mogul et al. [21] measure that current generation switches generate about 500 control messages per second, and as mentioned above, current single-threaded controllers can handle three orders of magnitude more messages per second.

Furthermore, we expect most controllers to be compatible with this concurrency model. Our model imposes only the modest requirement that the control algorithm should be able to process messages from different switches as they arrive, without depending on knowing the exact ordering of the events generating those messages if the ordering of those events is in fact nondeterministic.

McNettle therefore enables developers to write scalable control applications using the powerful computational capabilites and convenient abstractions available on multicore servers. Table 2 lists several example control applications, their key features and requirements, and the support that McNettle provides to meet these requirements. In particular, low latency shared memory allows us to track network state at flow arrival rates, allowing us to support systems, such as Hedera [5], which performs visible updates of global network state on every flow arrival event in order to load balance traffic in the network.

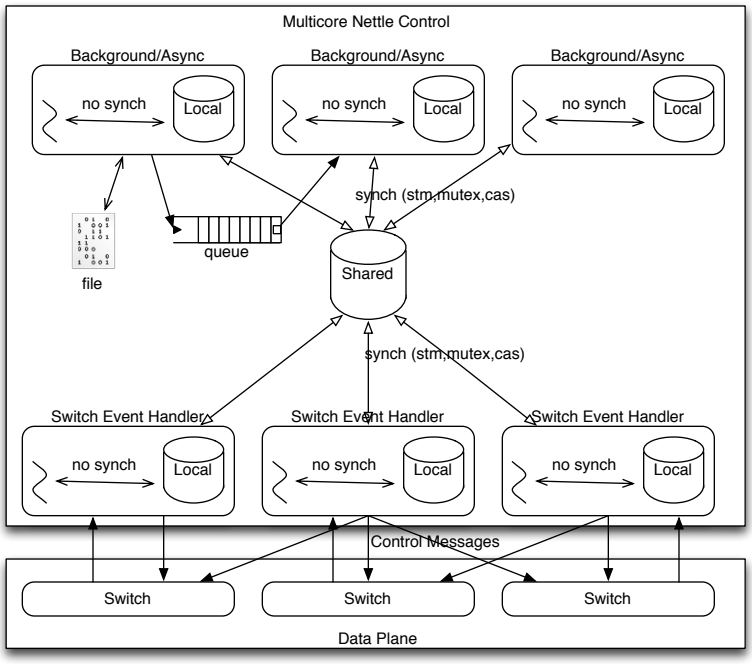

Figure 3: Architecture of McNettle controllers.

4 Programming Model

McNettle's programming model allows users to write control programs triggered by switch-initiated events as well as control programs that execute independently of switch events which initiate interaction with switches by sending commands. The programming model consists of the following components, illustrated in Figure 3:

- *switch event handlers*, each running concurrently, repeatedly and sequentially processing messages from a particular switch. Switch event handlers may respond to their particular switch, and may send commands to other switches.
- Background or *asynchronous threads*, which may access state shared with switch event handlers or with other asynchronous threads. These threads may send messages to switches and may perform IO with non-Openflow devices, such as files, network devices, or user interfaces.
- *switch-local state* that can be accessed only by a switch event handler. Switch local state can be accessed without synchronization.
- *network state* that can be accessed by any switch event handler or asynchronous thread and is manipulated using various techniques provided in Haskell, including software transactional memory (STM), mutex based synchronization, and non-blocking synchronization.

McNettle is implemented in Haskell, and we will introduce features of Haskell as necessary throughout this section. A Haskell program, like a C program, consists of a collection of definitions, and a single $main$ function. In Haskell, function application is written without parentheses, e.g. we write $f\ x$ instead of $f\ (x)$. More arguments can be provided as well; for example $g\ x\ y$ applies g to two arguments.

$runServer\ params\ hfact$	start the server with parameters $params$
$closeServer\ server$	closes connections
$defaultHandler$	default switch event handler
$forward\ pkt\ action$	forward a packet with a forwarding action.
$forwardWithRule\ pkt\ match\ act\ opts$	install a rule and forward the packet
$installRule\ sid\ match\ act\ opts$	install a flow table rule in a switch.
$deleteRules\ sid\ match$	deletes rules matching a given condition
$\mathbf{do}\ \{\, cmd1; ...; cmdn \,\}$	sequential composition
$PacketIn\ \{\, etherSrc, etherDst, ... \,\}$	The packet-in record consisting of ethernet header fields.
$exactMatch\ pkt$	A match on every header field of the packet.

Figure 4: McNettle Server API.

Figure 4 lists some key parts of the McNettle API. The API provides methods to start and stop the McNettle server. When $runServer$ is executed, the server opens a TCP listen socket on which it waits for Openflow switches to connect. When a switch connects, the server exchanges initial messages with the switch to obtain basic switch configuration, including a unique switch identifier (sid). The server then invokes the switch handler factory (specified as the second argument to $runServer$) with a record containing the basic configuration information obtained.

A switch event handler is a record of handlers, one for each of the various which a switch may send to the controller. We provide a basic switch handler, denoted $defaultHandler$, that has default definitions which simply return without performing any actions. A non-default handler can then be created by specifying just the handlers of interest. The following demonstrates a complete but minimal "hello world" controller:

$main = runServer\ defaultParams\ factory$
$factory\ switchConf =$
  $return\ (defaultHandler\ \{\, onPacket = \lambda pkt \rightarrow forward\ pkt\ allPorts \,\})$

The main function simply starts the server with default parameters and the specified switch handler factory $factory$. $factory$ returns a handler that overrides the default handler's method for handling packet-in messages. The packet-in handler above instructs the switch to forward the packet out of all ports (excluding the incoming port of the packet). Other forwarding actions, such as forwarding to a particular port are also possible. The $forwardRule$ function instructs the switch first to install a rule and then to reprocess the packet. Other commands like $installRule$ and $deleteRule$ manipulate switch tables. These, and other IO actions, can be combined in sequence using Haskell's do-notation, as we will illustrate with examples in the next section.

McNettle's library includes a number of algorithms and data structures, such as various concurrent hash tables and single and all-pairs routing algorithms. These libraries use a variety of Haskell's features, such as $IORef$, $MVar$ and $TVar$ variables to implement data structures with specific concurrent behaviors.

We now describe several simple network controllers, and illustrate how they make use of the features provided in McNettle.

```
factory conf =
  do table ← newHashTable size hashfun
     let ph = packetHandler table
     return (switchHandler { onPacket = ph })
packetHandler table pkt =
  do insert table (etherSc pkt) (inPort pkt)
     result ← lookup table (etherDst pkt)
     case result of
       Nothing        → forward pkt flood
       Just destPort → forwardRule pkt (exactMatch pkt) (phyPort destPort) opts
```

Figure 5: McNettle Local Learning Controller

4.1 Local Learning Controller

As a first example, we consider a simple controller that performs a function similar to a "learning switch". In this controller, each switch handler examines packets arriving at its switch to infer in which direction to send packets for each host in the network. In more detail, each switch handler builds a map (associative dictionary), associating a MAC address with the switch port on which a packet from that address was last seen. When a packet arrives at a switch and generates a packet-in message, the switch handler looks up the packet's destination MAC address in the map. If it finds a port, it sends the packet to that port and otherwise forwards the packet out of all ports (except the incoming port).

Figure 5 shows the code for the switch handler factory and the switch handler. The handler factory creates a new hash table for the switch handler and then returns a handler that overrides the packet-in handler function and has access to the newly created hash table. The packet-in procedure updates the hash table entry with the sender's MAC address, then looks up the destination in the hash table. It then performs a case analysis on the result, which is needed because the resulting value indicates whether the lookup succeeded and if so what the value of the lookup is, or fails. The hash table requires an initial size and a hash function on MAC addresses, whose definitions we omit here. The factory initializes a new hash table for every switch and therefore switches do not share any state in this controller.

This controller mimics the traditional network model in which switches have no shared state and demonstrates how our design allows efficient synchronization-free concurrency when sharing is not needed.

4.2 Global Learning Controller

The previous local learning switch has the drawback that one switch thread may be unaware of a host's location even though some other switch thread has learned the host's location. This may cause the controller to flood more packets than it would if the switch threads shared their knowledge about host locations. To remedy this, we write a second controller that uses a globally visible host location table. Since this table is accessed concurrently we must use some synchronization methods to correctly update and query it. In Section 6 we will evaluate several of alternatives synchronizations available to us in Haskell.

4.3 Shortest Path Routing Controller

The shortest paths routing controller maintains several global variables, including a variable referring to a topology data structure and a table of variables, one per switch, referring to a switch's next-hop table. In addition, it must recompute the routing tables of every switch when the topology changes. By using an parallel single source routing algorithm, like Dijkstra's algorithm, we can take advantage of

```
reserveLink linkVars amt u v =
  do let var = linkVariable linkVars (u, v)
     current ← readTVar var
     when (current < amt) retry
     writeTVar var (current − amt)
reservePath linkVars path amt
  = res path `orElse` return False
  where res [ ]          = return True
        res (u : [ ])    = return True
        res (u : v : rest) = do {reserveLink linkVars amt u v; res (v : rest)}
```

Figure 6: Bandwidth reservation using STM.

Haskell's extensive support for parallel evaluation of functions. For example, having implemented a function *dijkstraRoutingTable*, given the topology *topo* and a list of switches *sws* we can evaluate the new routing tables in parallel with this one-liner:

parMap (*dijkstraRouting topo*) *sws*

4.4 Bandwidth-on-Demand Controller

As a final example, we consider a bandwidth-on-demand controller, which illustrates a controller that performs a complex state update during flow request processing.

At a high-level our controller will work as follows: it will maintain a global table of host locations, a global shared map of the available bandwidth on each network link, and route tables. When a packet-in arrives from a switch, the receiving switch thread looks up the destination location and retrieves a route to this location. It then attempts to reserve bandwidth on each link along the route. The reservation may fail if granting the reservation would cause oversubscription on any link in the path. If the reservation succeeds, flow rules are installed in switches and the packet is sent. Otherwise, the packet is dropped.

Figure 6 demonstrates the use of Haskell's STM to structure these reservations as transactions and illustrates the use of alternatives, conditional retries, and nesting of transactions. The reservation program is a clear transcription of the high-level requirements of the controller and what is missing is just as important as what is present: no locking, and no need to compensate for failing reservations. Finally we can execute a transaction with:

atomically (*reservePath linkVars path amount*)

5 Implementation

We build McNettle in Haskell using the Glasgow Haskell Compiler (GHC) and runtime system (RTS), leveraging its extensive support for multicore execution. Throughout this section we refer to experiments which demonstrate the efficacy of our implementation techniques. We will explain the details of our experimental setup in section 6.

5.1 Scheduling Event Handlers

The heart of our system is the scheduler, illustrated in Figure 7, which is a lightly modified version of GHC's scheduler. The GHC runtime system (RTS) implements application level threads, called *Haskell threads*. The GHC RTS schedules these threads over a set of processor cores using a set of OS kernel threads, roughly one kernel thread per processor. Haskell threads are lightweight [20] and the RTS can switch between them

without incurring the cost of an OS context switch. Each worker thread maintains a run queue consisting of runnable Haskell threads, and services these Haskell threads in round robin fashion. Each thread runs until it yields cooperatively or preemptively due to its time slice expiring, or until it blocks on a mutex or otherwise terminates, and in these latter cases it is removed from the run queue. RTS threads periodically load balance Haskell threads among each other by sharing their run queues with idle RTS threads.

We implement switch event handlers as Haskell threads, called *switch threads*. Each switch thread repeatedly reads from its OpenFlow switch socket, parses messages, runs user-specified control logic, and then sends generated messages to switch sockets. The $recv$ and $send$ system calls used to transfer data to switch sockets are performed in non-blocking mode. If they fail because they would block, the switch thread registers its interest in the socket file descriptor with appropriate IO Managers (described below) and block. With this design, we process a single switch's messages in order, but process distinct switches' messages concurrently. Furthermore, the load balancing scheduler ensures that processor capacity is shared approximately fairly among active switch handlers.

This design ensures that each OpenFlow message is processed by a single core. We explored alternative approaches, in particular pipelining operations on packets with different portions of the pipeline on different cores as well. Our experiments were consistent with the findings of Dobrescu [11], namely that processing each packet on a single core outperforms pipelining, due to the burden of bringing packet data into the caches of several cores, rather than just once, and the overhead of core synchronization required to wait and signal when pipeline inputs are ready.

In order to reach our desired latency, we slightly modify GHC's scheduler and use an application-specific IO manager. In particular, we extend the GHC RTS by adding a pool of dedicated IO manager RTS threads, each executing exclusively on a core. An IO manager thread uses $epoll$ to monitor a set of switch sockets file descriptors and uses a *callback table* to lookup callback functions when a file descriptor is ready to perform IO. As mentioned above, switch threads which need to block waiting for IO instead register a callback with the callback table, add their file descriptor to an epoll object and then block.

By reserving cores exclusively for IO managers, we ensure that an IO manager thread cannot be delayed by some other RTS thread, for example one running a switch event handler. This is crucial in order to dispatch work quickly to idle RTS threads. In contrast, in the GHC RTS a single IO manager is scheduled in the same way as all other application level threads and may incur delays in executing while waiting for another RTS thread to release a core.

Futhermore, we take advantage of application-specific features to implement an application-specific IO manager that outperforms the general-purpose IO manager in GHC. The most important improvement we make takes advantage of the knowledge that each switch thread accesses a distinct file descriptor. With this constraint, all registration commands commute and can proceed concurrently with no synchronization. In contrast the GHC IO manager uses a single lock around the callback registration table, which becomes highly contended in our workloads. In particular, this lock can prevent the IO manager from dispatching work, while other worker threads take the lock to register a callback.

Figure 8 shows the mean response time of the local learning controller with several variants. The first variant is GHC's standard IO manager. The second uses GHC's standard manager with its single callback table lock divided into 32 locks. The third variant uses a single instance of our application-specific IO manager, scheduled as an ordinary Haskell thread. The fourth variant reserves a core for our single application-specific IO manager and the last uses two dedicated cores and application-specific IO managers. Each run is performed using 16 total cores and with each switch sending 1000 messages per second. The results show that each improvement makes a substantial improvement to the latency, with the cumulative improvement decreasing latency by over a factor of 30 in the case of 100 switches. We also see that we

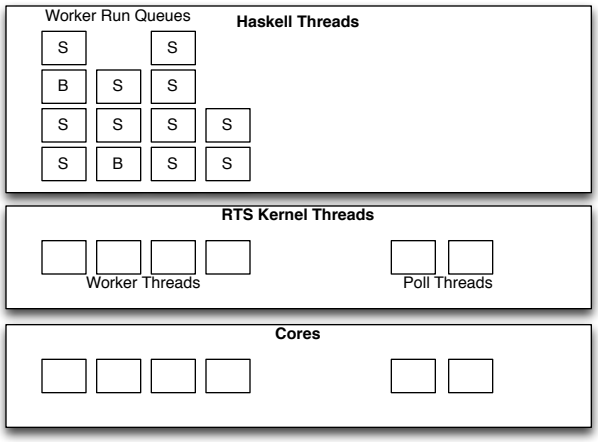

Figure 7: Implementation of McNettle scheduler.

	mean latency (milliseconds)	
Variant	100 switches	500 switches
GHC mgr	10.63	29.82
GHC mgr, striped lock	2.52	16.94
1 lockfree mgr, 0 IO cores	2.43	5.25
1 lockfree mgr, 1 IO core	0.81	5.68
2 lockfree mgrs, 2 IO cores	0.32	3.29
ping	0.2	0.2

Figure 8: Latency of McNettle running the local learning controller using different IO manager designs. We present the results with workload of 100 switches and 500 switches. We also show the mean ping latency between our the server and switch simulation machines.

are able to achieve under 1ms latency with 100 switches sending at a rate exceeding the maximum rate commercially available OpenFlow switches.

A final scheduling challenge is to schedule the outgoing messages of switch handlers. This problem arises when multiple concurrent switch handlers write to a single switch. This can arise in controllers that provision a flow across the whole network in response to a message received by a single switch. Because Openflow messages are serialized to a switch over a single TCP stream, the concurrent handlers might interleave bytes of their messages if they both write to the socket concurrently. We use a mutex to protect access to each outgoing socket; blocked threads form a queue waiting to send. While this could lead to diminished throughput as in head-of-line blocking in packet switches, we expect there to be low contention on outgoing sockets, in which case this queuing will not significantly diminish performance.

5.2 Message Processing

By processing messages from each switch's TCP stream in a separate thread, we are able to optimize message parsing dramatically. Switches send variable-length, but typically short — on the order of 100 bytes — control messages over the TCP stream, with the length included in an application-layer header field. Reading a single message at a time from the TCP socket requires 2 system calls: one to read the short

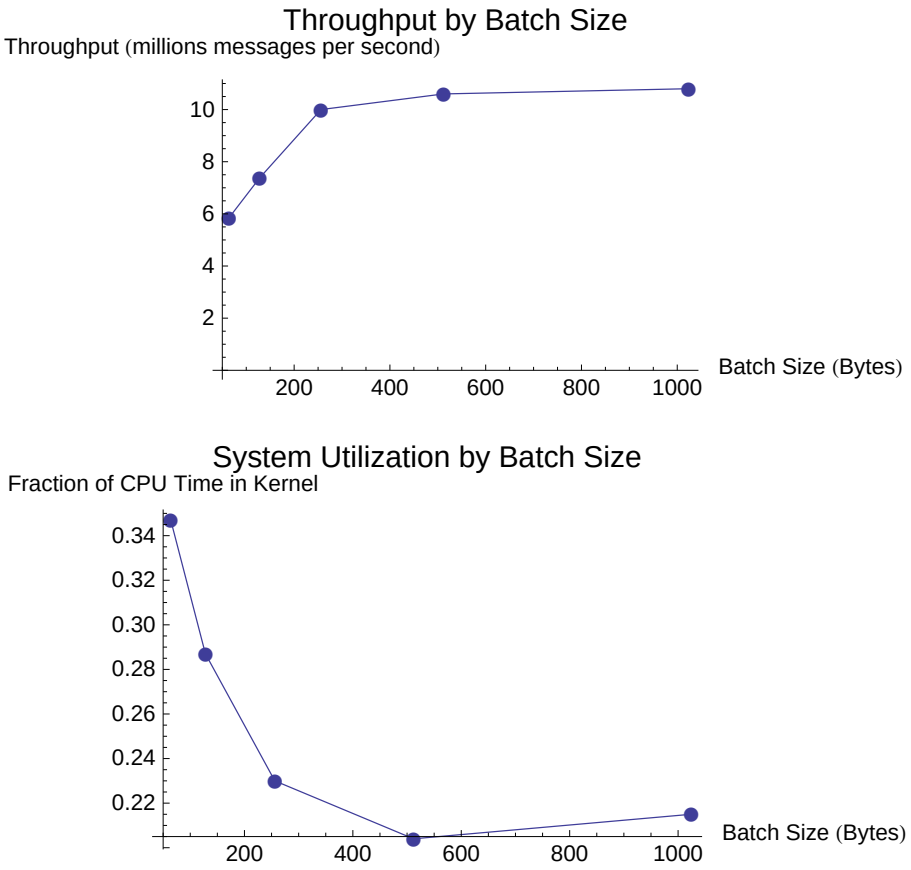

Figure 9: Maximum throughput (in millions of messages per second) and percent system time (of total CPU time) as a function of batch size when running the local learning controller with 40 cores and 500 switches.

fixed length header and another to read the remaining body of the message. This becomes a bottleneck at high throughput rates. To reduce the overhead, we read from the socket in chunks and process a number of messages in the chunk. We typically have some left-over bytes at the end of the chunk which make up a message fragment, and this must be retained when processing the next chunk. This reduces the number of system calls and dramatically improves performance. Note that this optimization would be impossible if a switch's socket could be accessed concurrently, because of the possibility of reading partway in a message fragment. Figure 9 demonstrates the throughput and proportion of execution time in system calls for the learning switch controller when each message is received separately and when batching is used with various batch sizes.

We apply the same technique to output: we run the user-specified control logic on several messages, serializing the outgoing messages into a buffer which we then write to the socket after processing a single batch of incoming messages.

We expect some OpenFlow messages occur more frequently than others. In particular, flow table modifications that include an exact match occur more frequently than others. We provide two commands, one for exact matches and one for the general case, which allows the implementation of the exact match command to optimize serialization by avoiding having to check which parts of the header are wildcards.

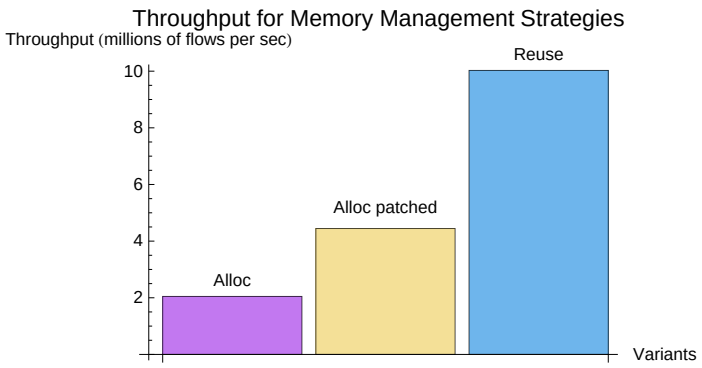

Figure 10: Maximum throughput (in millions of messages per second) for different memory management strategies when running the local learning controller with 40 cores and 500 switches.

We inline key functions and make data types and function arguments strict throughout the parsing and serialization library. Inlining helps by reducing the number of procedure calls and by providing the compiler more opportunities to apply optimizations. Strictness in data type definitions eliminates many pointers and instead unpacks fields directly into the parent data type. Strictness in function pointers and data type definitions avoids many boxing and unboxing operations and pointer dereferences.

5.3 Memory Management

Since we allocate data when parsing data for and running switch event handlers, our allocation rate is proportional to the system throughput. Furthermore, the commonly used libraries in Haskell for reading from (and writing data to) sockets use common functional programming idioms: they allocate a new immutable byte array of the desired size and fill it with data from the socket buffer. This immutable byte array is then passed to parsing functions. For our application, with throughput around 10Gbps, these byte arrays alone would require allocating over 1 GBps.

Our work uncovered a scaling problem in the way GHC's allocator allocated byte arrays. GHC's allocates byte arrays from a global block pool, rather than the core-local nurseries that are used for other objects, and this global block pool is protected by a lock. This lock became heavily contended by our programs. The GHC maintainers have since addressed this problem by allocating byte arrays using the same technique as for other objects.

Despite fixing the problem of byte array allocation in GHC, we found that performance improved dramatically by allocating fixed size areas of memory for each switch thread to read socket data into and to serialize data out, rather than than allocate memory to read each batch of switch messages. Figure 10 shows the maximum throughput for the local learning controller running with 40 cores and 500 switches when allocating a new byte array for every socket read, both with the bottleneck we observed in GHC's allocator, and with the proposed fix to this problem, and when reusing a byte array as a buffer.

GHC's RTS uses a parallel, generational, stop-the-world garbage collector [18]. While it performs well and is highly parallelized, the stop-the-world collection requires all Haskell RTS threads to synchronize with barriers to start garbage collection and to start mutators. This all-core synchronization can become a scalability problem. We therefore prefer to use a large allocation area (generation 0 area) to reduce the frequency of garbage collections. Figure 11 demonstrates the effect of different allocation area sizes and numbers of cores on the maximum throughput and average pause times of the local learning controller.

As mentioned, GHC's garbage collector is not concurrent; that is, all cores must stop executing their mutators in order to execute garbage collection. Therefore, pause times may become a serious problem for

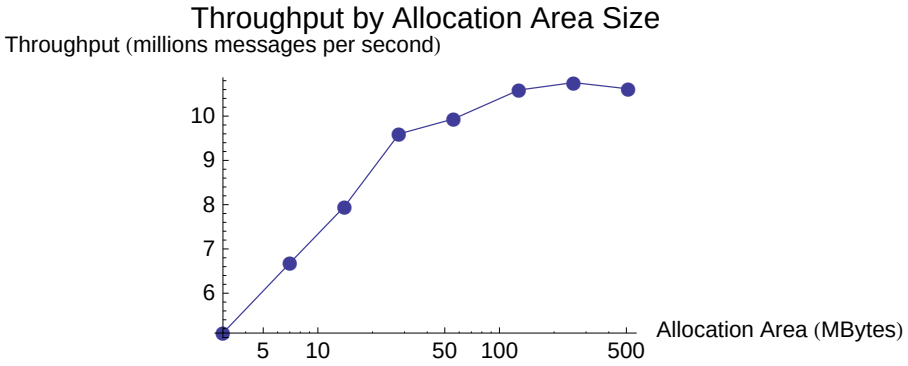

Figure 11: Maximum throughput (in millions of messages per second) for different allocation areas (generation 0) when running the local learning controller with 40 cores and 500 switches. The horizontal axis is logarithmic.

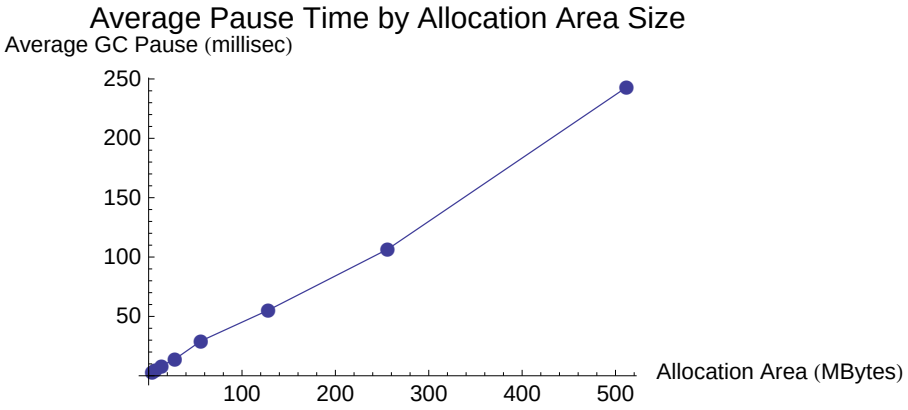

Figure 12: Average GC pause time for different allocation areas (generation 0) when running the local learning controller with 40 cores and 500 switches.

controller responsiveness. Unfortunately, pause times increase as the allocation are increases, as measured in Figure 12. Recent efforts have been made to implement a thread-local collection algorithm [19] for GHC, which would allow cores to perform core-local garbage collection without stopping all other cores, but this has not been incorporated into GHC yet.

5.4 Interrupt Handling

At our target message rates of over 10 million OpenFlow messages per second, we require more than 10 Gbps network IO bandwidth to our server. While technologies such as Message Signaled Interrupts (MSI and MSI-X), multi-queue NICs and Direct Cache Access (DCA), have been adopted by modern multicore processors and operating systems, it is still difficult to receive and process small packets at 10Gbps when a 10Gbps NIC is used.

In particular, we experienced scaling limitations due to hardware interrupt affinity. Linux commonly uses *irqbalance* [4] to automatically adjust the hardware affinity of peripheral equipments, including NICs. *Irqbalance*, however, tries to evenly distribute workloads from a NIC to CPU cores, in contrast to guaranteeing packets of the same flow being handled by the same CPU core, which might incur severe cache thrashing [24]. Moreover, in experiments, we found that binding each queue of a multi-queue NIC to the

same numbered cores (e.g binding queue 0 to core 0, binding queue 1 to core 1, etc.) increased system performance by up to 20% over using irqbalance to binding queues to CPU cores.

6 Evaluation

In this section, we evaluate McNettle, measuring how throughput scales with network size and core counts. We evaluate fairness and compare the scaling of McNettle with two other multi-threaded OpenFlow controllers. Finally we compare different methods for implementing shared memory concurrent hash tables and demonstrate the feasibility of updates at flow rates in excess of 10 million flows per second.

6.1 Server Architecture

We use a DELL Poweredge R815 server with 48 cores and 64 GB memory with Broadcom NetXtreme II network adapters with 8 1Gbps Ethernet ports, and one two port Intel 10Gb NIC with an 82529 Ethernet controller. The 48 cores are provided by four AMD Opteron 6164 processors [9]. Each processor holds two dies, each die containing six cores sharing a 6M L3 cache. The cores, dies and RAM banks form a hierarchy with significant differences in available memory bandwidth and latency due to the location of cores and memory.

6.2 Workload and Measurement

We simulate switches using a slightly modified version of the *cbench* [8] tool, a publicly available, open-source OpenFlow controller benchmarking program, which we call *ycbench*. ycbench opens a TCP connection to the controller for each switch being simulated, performs an initial handshake and then sends packet-miss messages and monitors controller responses. It performs several rounds of tests, and both the number of rounds and duration of each round is configurable.

ycbench can generate workloads of three different types. The max-rate mode consists of each switch sending control messages as fast as possible, i.e. as long as its sending buffer has sufficient space. In ping-pong mode, each switch begins by sending a message, and then sends one message every time it receives a response. In fixed-rate mode, each switch sends at a fixed message rate. ycbench measures throughput by counting the number of responses each switch receives to its packet-miss events and estimates latency by having each switch record the sending time and response time for a fraction of packets.

Unless otherwise noted, we run ycbench on a collection of 9 test machines, with 9 1Gbps and 2 10Gbps interfaces all linked through an HP switch with 1Gbps and 10Gbps port modules, terminating at the control server with 8 1Gbps and 2 10Gbps ports. The number of switches to be simulated in a given test is divided up among the test machines according to the amount of network bandwidth each test machine has to the server.

6.3 Throughput Scaling

Figure 13 demonstrates throughtput scaling with the number of cores through 46 cores, for the local learning controller with different numbers of switches, up to 5000 switches. Throughput reaches nearly 14 million flows per second. We also see that aggregate throughput is lower for larger number of switches. We believe this is explained by considering that the scheduler is multiplexing more distinct threads over the fixed number of cores. There is some scheduler overhead and loss of locality as each thread switch requires different memory to be in context.

6.4 Maintaining Low Latency

We test latency using ycbench in ping-pong mode. By adding switches we increase the load, since each switch is independently performing ping pong. For loads from 1-100 switches using 5-40 cores we observe

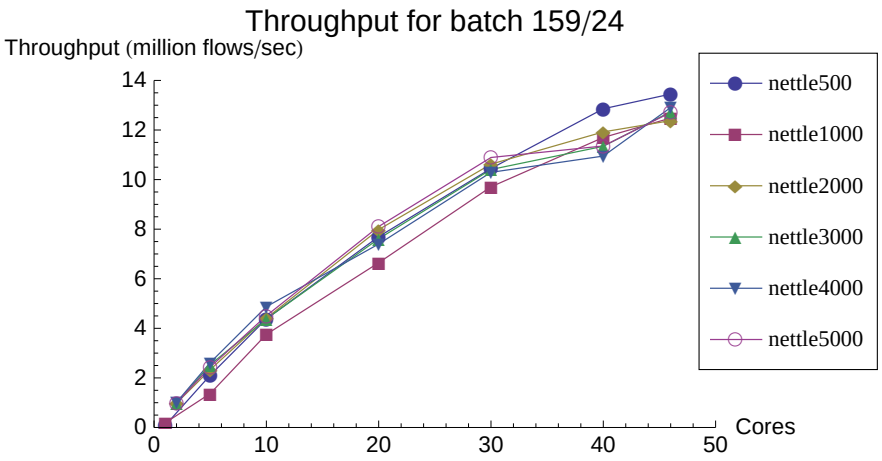

Figure 13: System throughput for the McNettle local learning controller as a function of number of cores, when run with number of switches varying from 500 to 5000.

latency of under 500 microseconds. Figures 14 and 15 show the latency as the number of switches (and consequently the load) increases from 100 to 2000 switches, using different numbers of cores. In Figure 14 we see that using more than 5 cores dramatically reduces the latency when the number of switches grows beyond 500 and in Figure 15, which simply omits the 5-core curve, demonstrates that increasing cores beyond 10 again substantially reduces latency with a load of more than 500 switches. Furthermore, we see that with 15 or more cores, McNettle is able to handle 1500 switches with under 10 ms latency.

6.5 Fairness

With a ping-pong load and the local learning switch, McNettle achieves a Jain fairness of over 0.996 for all measured numbers of cores from 5-46 and for all measured numbers of switches, from 100-2000.

On the other hand, for the max rate workload, the Jain fairness of McNettle is not as good. For example, Figure 16 shows the fairness as a function of the number of switches for several different numbers of cores. The fairness is persistenly well below 1, and sometimes close to 0.5. We believe that this is due to a limitation of the GHC run time system's thread scheduler, which allows for persistent unfairness under some workloads. In particular, GHC only load balances threads by moving some core's threads to another core when one core notices that another one is idle. Therefore, it can occur that some cores have much fewer threads than other cores, and if all of the threads are constantly runnable, no core will be idle, and hence the load balancer will never migrate threads. Unfortunately, the max rate workload is exactly the type of workload which creates this scenario, since each switch thread almost always has control messages to process. This limitation could be overcome by incorporating fair scheduling (e.g. [17]) into the GHC runtime system. On the other hand, such workloads may not arise frequently under realistic conditions.

6.6 Comparison with NOX and Beacon

In this experiment, we evaluate the throughput performance and scalability of the learning controllers written in McNettle, NOX, and Beacon for networks of up to 5000 switches. For each network size we evaluate the

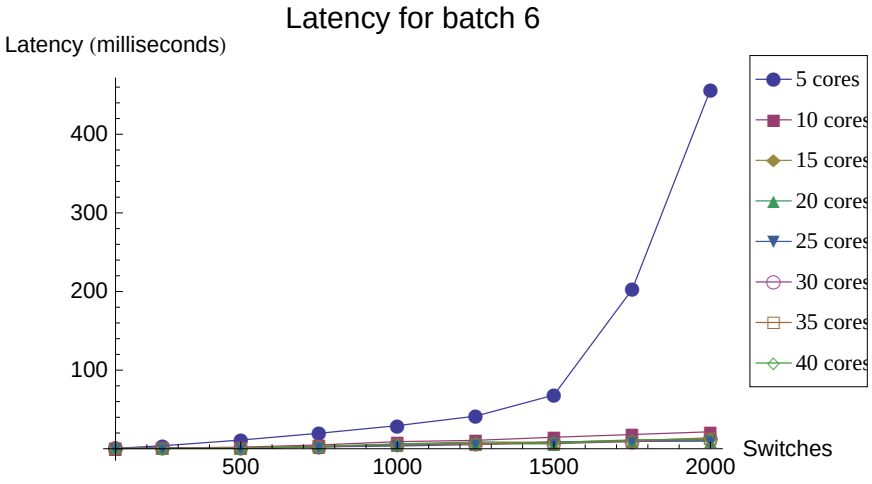

Figure 14: Latency for the McNettle local learning controller as a function of number of switches, with different numbers of cores.

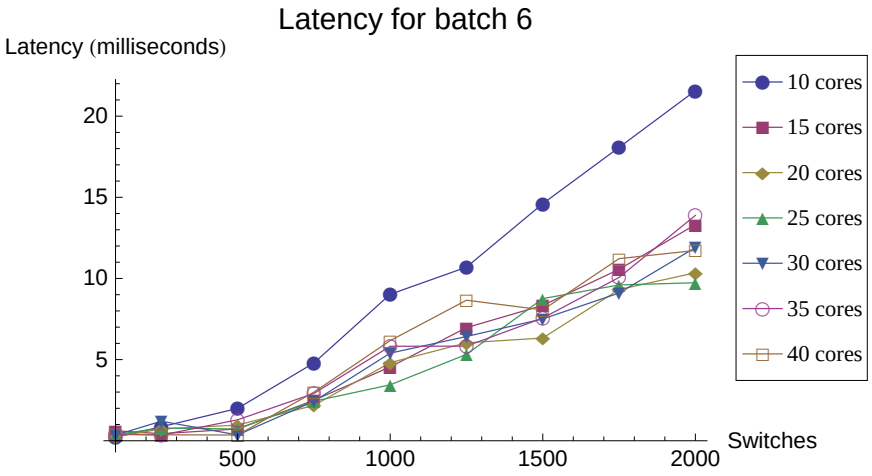

Figure 15: Latency for the McNettle local learning controller as a function of number of switches, with different numbers of cores.

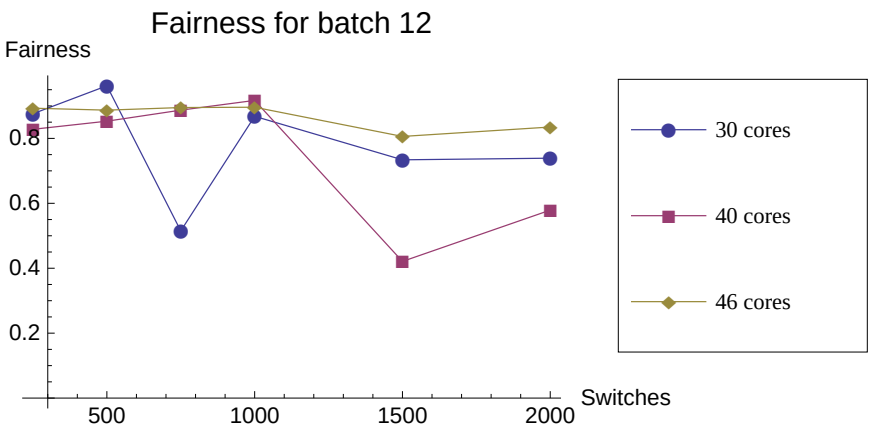

Figure 16: Jain fairness for the local learning controller with the max-rate workload as a function of number of switches for several different numbers of cores.

throughput for increasing number of cores. We use the multi-threaded version of NOX[1] and Beacon v1.0.0 2011-09-12. We ran all controllers with CPU affinity on. In all cases Beacon was run with the recommended benchmarking settings [2].

Figure 17 shows maximum throughput scaling results. We see that the McNettle controller outperforms the NOX controller for all loads with performance up to over six times of NOX. We also see that the McNettle controllers scale up to through 45 cores, while the NOX controllers scale to about 10 cores. Beacon scales well through 20 cores, but stops scaling for higher core counts. Beacon obtains better throughput than McNettle for less than 30 cores, but lower peak throughput.

Figure 18 shows the latency of McNettle and the latency of NOX for 100, 250, and 500 switches with both 15 and 20 cores. We see that McNettle obtains dramatically better latency. Beacon performs much better than NOX with respect to latency, maintaining low latency with 20 cores up to 1000 switches. Above 1000 switches, Beacon's response time begins to grow dramatically. The response times of Beacon and McNettle under the ping-pong workload for up to 2000 switches is shown in Figure 19. We see that Nettle maintains low latency (under 14 ms for all numbers of cores tested) while Beacon's latency grows up 25-70 milliseconds at 2000 switches.

6.7 Cost of Shared State

To evaluate the effects of reading and mutating shared state within the event handlers, we measure the throughput scaling for the global learning controller using several different shared state mechanisms in Haskell. We implement the same algorithm, varying only the method of ensuring consistent update of the global host location table. In the first variant, we use Haskell's $IORef$ variables in the underlying hash table and use atomic compare-and-swap instructions to update the table without blocking. In the second variant we use Haskell's $MVar$ datatype, which essentially pairs a variable with a mutex protecting it and exposes an API that requires taking the mutex before reading or writing the variable. In the final version, we use Haskell's STM system to perform the updates. Figure 20 demonstrates that the $IORef$-based mutable hash table scales as well as the local learning switch, while the $MVar$ and STM variants perform worse. The

[1] git commit e9c3da6bb12ad3fa0d2b609e697a50ce44ca19f4

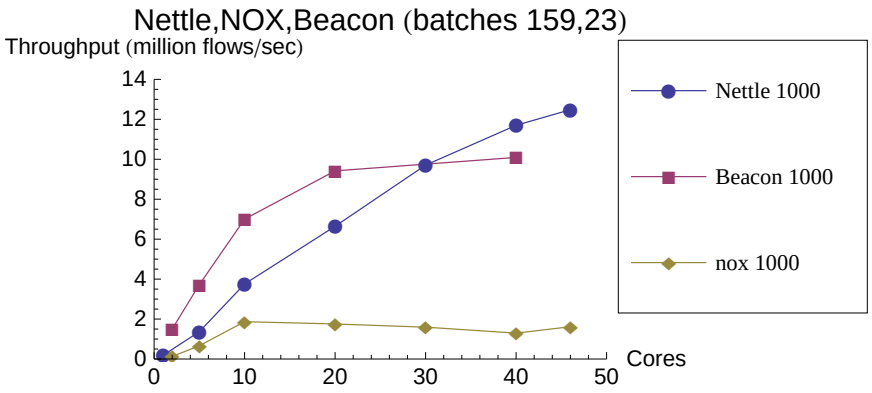

Figure 17: System throughput for both the McNettle and NOX learning controllers as a function of the number of cores used by the controller for varying loads corresponding to different network sizes as represented by the number of switches controlled by the controller.

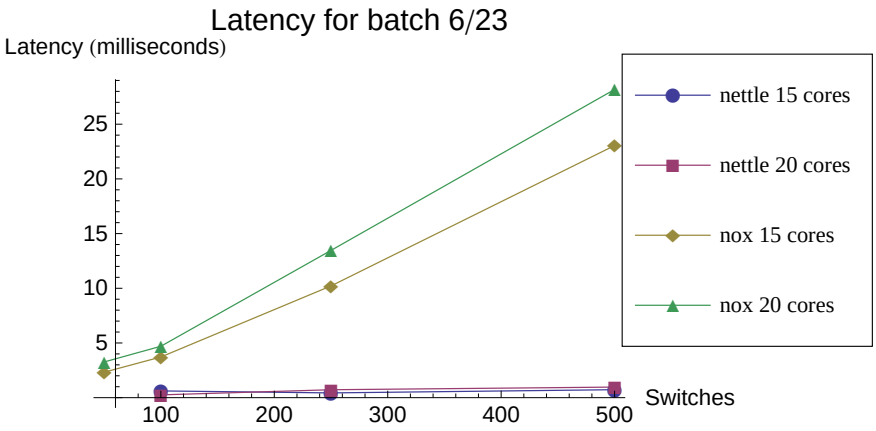

Figure 18: Latency for both the McNettle and NOX learning controllers as a function of the number of switches executing ping-pong workload for different numbers of cores.

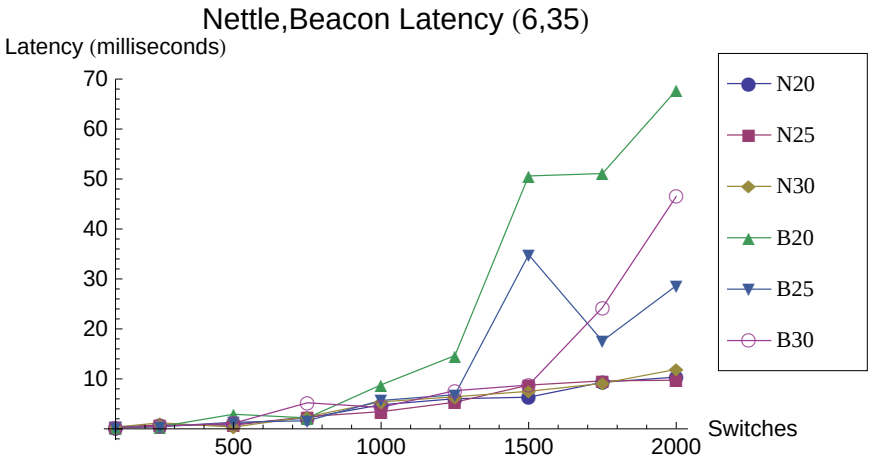

Figure 19: Latency for both the McNettle and Beacon learning controllers as a function of the number of switches executing ping-pong workload for different numbers of cores.

$MVar$ variant requires hash table reads to take locks, causing unecessary queueing while STM incurs a higher overhead and may incur transaction restarts.

This experiment demonstrates the need for low-overhead synchronization techniques for critical data structures. These techniques have a higher cost in terms of programmer effort, but are needed to obtain optimal performance. On the other hand this experiment demonstrates that STM transactions can be sustained at rates of several million transactions per second, indicating that this method can be used for more complex state manipulations that occur on a smaller fraction of flow requests.

7 Related Work

NOX [13] is the original OpenFlow implementation. Although single-threaded by default, it has an experimental branch, called NOX-destiny, which uses the Boost [3] libraries to for IO and threading. Their concurrency model differs from McNettle's in that messages from a single switch can be processed out of order. As a result, they do not provide a notion of switch-local state that can be accessed without synchronization.

Beacon [1] is a multi-threaded Java controller framework that performs well for a modest number of cores. Our system differs in several ways. Unlike Beacon, we explicitly target scalability to large numbers of cores and thousands of switches. We also emphasize the use of high-level programming methods, from functional programming for expressing algorithms to software transactional memory for expressing complex state updates.

Maestro [7] is another OpenFlow implementation for scaling across multiple cores. It provides automatic parallelism for single-threaded OpenFlow controllers using techniques such as batching, pull-based work dispatch, and cache optimization for multiple cores. The main cost is that Maestro's pipeline is relatively fixed, consisting of a few well-defined OpenFlow tasks, rather than a fully extensible programming system like McNettle.

Several high level languages and systems have been proposed for OpenFlow programming and configuration, including FML [15], Frenetic [12], and Nettle [25]. These languages focus on higher-level

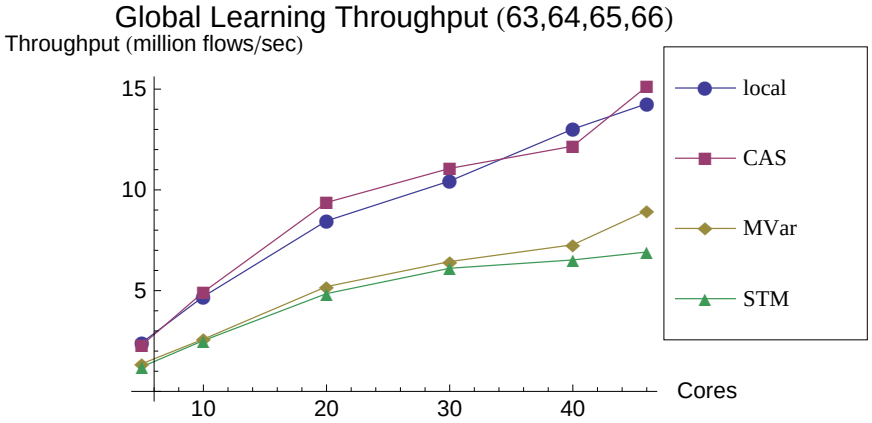

Figure 20: Throughput for the McNettle local learning controller and several global learning controllers as a function of the number of cores used. All runs are made with 1000 switches and max rate workload.

abstractions rather than multi-core performance. Similarly, Openflow API improvement efforts such as Devoflow [10] show how to refactor the OpenFlow API to reduce the coupling between centralized control and centralized visibility, so as to reduce the OpenFlow protocol costs. Onix [16] proposes a distributed control API for large-scale Software Defined Networks. McNettle is designed to explicitly incorporate such new APIs and controllers written to them.

Prior work has extensively focused on the use of emerging multicore architectures to accelerate network processing. RouteBricks [11] and PacketShader [24] aim to provide software routers, and MIDeA [14] presents an intrusion detection architecture for high speed networks. In contrast to prior work that mainly targets simple routing, encryption and intrusion detection, McNettle parallelizes a real network controller that involves complicated control plane and even shared state programming on multicore architectures

References

[1] `https://openflow.stanford.edu/display/Beacon/Home`.

[2] Beacon benchmarking guide. https://openflow.stanford.edu/display/Beacon/Benchmarking.

[3] Boost. [Online; accessed 02-May-2012].

[4] irqbalance. http://www.irqbalance.org/.

[5] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *In Proc. of Networked Systems Design and Implementation (NSDI) Symposium*, 2010.

[6] L. Bergstrom. Measuring numa effects with the stream benchmark. Technical report, University of Chicago, Chicago IL 60637, USA, 2011.

[7] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: Balancing fairness, latency and throughput in the openflow control plane. Technical report, Rice University, 2011.

[8] Cbench. Cbench, 2012. [Online; accessed 10-April-2012].

[9] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, Mar. 2010.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, 2011.

[11] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[13] 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, July 2008.

[14] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[15] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *WREN '09: Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 1–10, New York, NY, USA, 2009. ACM.

[16] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[17] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 65–74, New York, NY, USA, 2009. ACM.

[18] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[19] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 21–32, New York, NY, USA, 2011. ACM.

[20] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[21] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. Devoflow: cost-effective flow management for high performance enterprise networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM.

[22] OpenFlow. Openflow switch specification, version 1.0.0, 2011. [Online; accessed 6-April-2012].

[23] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the datacenter.

[24] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Midea: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[25] A. Voellmy and P. Hudak. Nettle: taking the sting out of programming network routers. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag.