High Performance Packet Processing with FlexNIC

Antoine Kaufmann[1] Simon Peter[2] Naveen Kr. Sharma[1]
Thomas Anderson[1] Arvind Krishnamurthy[1]

[1]University of Washington [2]The University of Texas at Austin
{antoinek,naveenks,tom,arvind}@cs.washington.edu simon@cs.utexas.edu

Abstract

The recent surge of network I/O performance has put enormous pressure on memory and software I/O processing subsystems. We argue that the primary reason for high memory and processing overheads is the inefficient use of these resources by current commodity network interface cards (NICs).

We propose FlexNIC, a flexible network DMA interface that can be used by operating systems and applications alike to reduce packet processing overheads. FlexNIC allows services to install packet processing rules into the NIC, which then executes simple operations on packets while exchanging them with host memory. Thus, our proposal moves some of the packet processing traditionally done in software to the NIC, where it can be done flexibly and at high speed.

We quantify the potential benefits of FlexNIC by emulating the proposed FlexNIC functionality with existing hardware or in software. We show that significant gains in application performance are possible, in terms of both latency and throughput, for several widely used applications, including a key-value store, a stream processing system, and an intrusion detection system.

Categories and Subject Descriptors C.2.1 [*Computer-Communication Networks*]: Network Architecture and Design

Keywords Network Interface Card; DMA; Flexible Network Processing; Match-and-Action Processing

1. Introduction

Data center network bandwidth is growing steadily: 10 Gbps Ethernet is widespread, 25 and 40 Gbps are gaining traction, while 100 Gbps is nearly available [47]. This trend is straining the server computation and memory capabilities; I/O processing is likely to limit future server performance. For example, last-level cache access latency in Intel Sandy Bridge processors is 15 ns [34] and has not improved in newer processors. A 40 Gbps interface can receive a cache-line sized (64B) packet close to every 12 ns. At that speed, OS kernel bypass [5; 39] is a necessity but not enough by itself. Even a single last-level cache access in the packet data handling path can prevent software from keeping up with arriving network traffic.

We claim that the primary reason for high memory and processing overheads is the inefficient use of these resources by current commodity network interface cards (NICs). NICs communicate with software by accessing data in server memory, either in DRAM or via a designated last-level cache (e.g., via DDIO [21], DCA [20], or TPH [38]). Packet descriptor queues instruct the NIC as to where in server memory it should place the next packet and from where to read the next arriving packet for transmission. Except for simple header splitting, no further modifications to packets are made and only basic distinctions are made among packet types. For example, it is possible to choose a virtual queue based on the TCP connection, but not, say, on the application key in a memcache lookup.

This design introduces overhead in several ways. For example, even if software is only interested in a portion of each packet, the current interface requires NICs to transfer packets in their entirety to host memory. No interface exists to steer packets to the right location in the memory hierarchy based on application-level information, causing extraneous cache traffic when a packet is not in the right cache or at the right level. Finally, network processing code often does repetitive work, such as to check packet headers, even when it is clear what to do in the common case.

The current approach to network I/O acceleration is to put fixed-function offload features into NICs [46]. While useful, these offloads bypass application and OS concerns and can thus only perform low-level functions, such as checksum processing and packet segmentation for commonly used, standard protocols (e.g., UDP and TCP). Higher-level offloads can constrain the system in other ways. For example, remote direct memory access (RDMA) [42], is difficult to adapt to many-to-one client-server communication mod-

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
ACM 978-1-4503-4091-5/16/04.
DOI: http://dx.doi.org/10.1145/2872362.2872367

els [32; 13], resulting in diminished benefits [25]. On some high-speed NICs, the need to pin physical memory resources prevents common server consolidation techniques, such as memory overcommit.

To address these shortcomings, we propose FlexNIC, a flexible DMA programming interface for network I/O. FlexNIC allows applications and OSes to install packet processing rules into the NIC, which instruct it to execute simple operations on packets while exchanging them with host memory. These rules allow applications and the OS to exert fine-grained control over how data is exchanged with the host, including where to put each portion of it. The FlexNIC programming model can improve packet processing performance while reducing memory system pressure at fast network speeds.

The idea is not far-fetched. Inexpensive top-of-rack OpenFlow switches support rule-based processing applied to every packet and are currently being enhanced to allow programmable transformations on packet fields [9]. Fully programmable NICs that can support packet processing also exist [35; 51; 10] and higher-level abstractions to program them are being proposed [22]. We build upon these ideas to provide a flexible, high-speed network I/O programming interface for server systems.

The value of our work is in helping guide next generation NIC hardware design. Is a more or less expressive programming model better? We are aiming to better understand the middle ground. At one extreme, we can offload the entire application onto a network processor, but development costs will be higher. At the other extreme, fixed-function offloads, such as Intel Flow Director, that match on only a small number of packet fields and cannot modify packets in flight are less expressive than FlexNIC, but forego some performance benefits, as we will show later.

This paper presents the FlexNIC programming model, which is expressive and yet allows for packet processing at line rates, and quantifies the potential performance benefits of flexible NIC packet steering and processing for a set of typical server applications. When compared to a high-performance user-level network stack, our prototype implementation achieves 2.3× better throughput for a real-time analytics platform modeled after Apache Storm, 60% better throughput for an intrusion detection system, and 60% better latency for a key-value store. Further, FlexNIC is versatile. Beyond the evaluated use cases, it is useful for high performance resource virtualization and isolation, fast failover, and can provide high-level consistency for RDMA operations. We discuss these further use cases in §6.

We make four specific contributions:

- We present the FlexNIC programming model that allows flexible I/O processing by exposing a match-and-action programming interface to applications and OSes (§2).
- We implement a prototype of FlexNIC via a combination of existing NIC hardware features and software emulation using dedicated processor cores on commodity multi-core computers and I/O device hardware (§2.4).
- We use our prototype to quantify the potential benefits of flexible network I/O for several widely used networked applications, including a key-value store, data stream processing, and intrusion detection (§3–5).
- We present a number of building blocks that are useful to software developers wishing to use FlexNIC beyond the use cases evaluated in this paper (§2.3).

Our paper is less concerned with a concrete hardware implementation of the FlexNIC model. While we provide a sketch of a potential implementation, we assume that the model can be implemented in an inexpensive way at high line-rates. Our assumption rests on the fact that OpenFlow switches have demonstrated this capability with minimum-sized packets at aggregate per-chip rates of over a half terabit per second [47].

2. FlexNIC Design and Implementation

A number of design goals guide the development of FlexNIC:

- **Flexibility.** We need a hardware programming model flexible enough to serve the offload requirements of network applications that change at software development timescales.
- **Line rate packet processing.** At the same time the model may not slow down network I/O by allowing arbitrary processing. We need to be able to support the line rates of tomorrow's network interfaces (at least 100 Gb/s).
- **Inexpensive hardware integration.** The required additional hardware has to be economical, such that it fits the pricing model of commodity NICs.
- **Minimal application modification.** While we assume that applications are designed for scalability and high-performance, we do not assume that large modifications to existing software are required to make use of FlexNIC.
- **OS Integration.** The design needs to support the protection and isolation guarantees provided at the OS, while allowing applications to install their own offloading primitives in a fast and flexible manner.

To provide the needed flexibility at fast line rates, we apply the reconfigurable match table (RMT) model recently proposed for flexible switching chips [8] to the NIC DMA interface. The RMT model processes packets through a sequence of match plus action (M+A) stages.

We make no assumptions about how the RMT model is implemented on the NIC. Firmware, FPGA, custom silicon, and network processor based approaches are all equally feasible, but incur different costs. However, we note that a typical commodity switch uses merchant silicon to support sixteen 40 Gbps links at a cost of about $10K per switch in volume, including the switching capacity, deep packet buffers,

protocol handling, and a match plus action (M+A) table for route control. In fact, it is generally believed that converting to the more flexible RMT model will *reduce* costs in the next generation of switches by reducing the need for specialized protocol processing [8]. As a result, we believe that adding line-rate FlexNIC support is both feasible and likely less expensive than adding full network processor or FPGA support.

In this section, we provide a design of FlexNIC, present a set of reusable building blocks implemented on FlexNIC, and describe how we emulate the proposed FlexNIC functionality with existing hardware or in software.

2.1 Motivating Example

To motivate FlexNIC and its design, we consider the performance bottlenecks for the popular Memcached key-value store [2] and how they could be alleviated with NIC support. Memcached is typically used to accelerate common web requests and provides a simple *get* and *put* interface to an associative array indexed by application-level keys. In these scenarios, service latency and throughput are of utmost importance.

Memory-efficient scaling. To scale request throughput, today's NICs offer receive-side scaling (RSS), an offload feature that distributes incoming packets to descriptor queues based on the client connection. Individual CPU cores are then assigned to each queue to scale performance with the number of cores. Additional mechanisms, such as Intel's FlowDirector [23], allow OSes to directly steer and migrate individual connections to specific queues. Linux does so based on the last local send operation of the connection, assuming the same core will send on the connection again.

Both approaches suffer from a number of performance drawbacks: (1) Hot items are likely to be accessed by multiple clients, reducing cache effectiveness by replicating these items in multiple CPU caches. (2) When a hot item is modified, it causes synchronization overhead and global cache invalidations. (3) Item access is not correlated with client connections, so connection-based steering is not going to help.

FlexNIC allows us to tailor these approaches to Memcached. Instead of assigning clients to server cores, we can *partition* the key space [30] and use a separate key space request queue per core. We can install rules that steer client requests to appropriate queues, based on a hash of the requested key in the packet. The hash can be computed by FlexNIC using the existing RSS hashing functionality. This approach maximizes cache utilization and minimizes cache coherence traffic. For skewed or hot items, we can use the NIC to balance the client load in a manner that suits both the application and the hardware (e.g., by dynamically repartitioning or by routing hot requests to two cores that share the same cache and hence benefit from low latency sharing).

Streamlined request processing. Even if most client requests arrive well-formed at the server and are of a common type—say, GET requests—network stacks and Memcached have to inspect each packet to determine where the client payload starts, to parse the client command, and to extract the request ID. This incurs extra memory and processing overhead as the NIC has to transfer the headers to the host just so that software can check and then discard them. Measurements using the Arrakis OS [39] showed that, assuming kernel bypass, network stack and application-level packet processing take half of the total server processing time for Memcached.

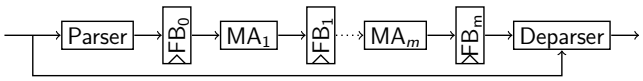

Figure 1. RMT switch pipeline.

With FlexNIC, we can check and discard Memcached headers directly on the NIC before any transfer takes place and eliminate the server processing latency. To do so, we install a rule that identifies GET requests and transfers only the client ID and requested key to a dedicated fast-path request queue for GET requests. If the packet is not well-formed, the NIC can detect this and instead transfer it in the traditional way to a slow-path queue for software processing. Further, to support various client hardware architectures, Memcached has to convert certain packet fields from network to host byte order. We can instruct the NIC to carry out these simple transformations for us, before transferring the packets into host memory.

2.2 FlexNIC Model

We now briefly describe the RMT model used in switches and then discuss how to adapt it to support application-specific optimizations of the type described above. We provide hardware implementation details where they impact the programming model.

RMT in switches. RMT switches can be programmed with a set of rules that match on various parts of the packet, and then apply data-driven modifications to it, all operating at line rate for the switched packets. This is implemented using two packet processing pipelines that are connected by a set of queues allowing for packets to be replicated and then modified separately. Such a pipeline is shown in Figure 1. A packet enters the pipeline through the parser, which identifies all relevant packet fields as described by the software-defined parse graph. It extracts the specified fields into a field buffer (FB_0) to be used in later processing stages. The relevant fields pass through the pipeline of M+A stages (MA_1..MA_m) and further field buffers (FB_1..FB_m). In a typical design, $m = 32$. An M+A stage matches on field buffer contents using a match table (of implementation-defined size), looking up a corresponding action, which is then applied as we forward to the next field buffer. Independent actions can be executed in parallel within one M+A stage. The deparser combines the modified fields with the

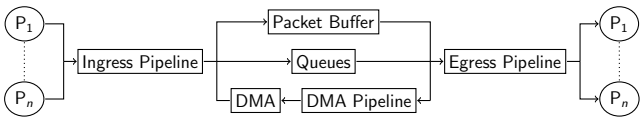

Figure 2. RMT-enhanced NIC DMA architecture.

original packet data received from the parser to get the final packet. To be able to operate at high line rates, multiple parser instances can be used. In addition to exact matches, RMT tables can also be configured to perform prefix, range, and wildcard matches. Finally, a limited amount of switch-internal SRAM can maintain state across packets and may be used while processing.

Applying RMT to NICs. To gain the largest benefit from our approach, we enhance commodity NIC DMA capabilities by integrating three RMT pipelines with the DMA engine, as shown in Figure 2. This allows FlexNIC to process incoming and outgoing packets independently, as well as send new packets directly from the NIC in response to host DMA interactions.

Similar to the switch model, incoming packets from any of the NIC's Ethernet ports ($P_1..P_n$) first traverse the ingress pipeline where they may be modified according to M+A rules. Packet fields can be added, stripped, or modified, potentially leaving a much smaller packet to be transferred to the host. Modified packets are stored in a NIC-internal packet buffer and pointers to the corresponding buffer positions are added to a number of NIC-internal holding queues depending on the final destination. In addition to host memory, hardware implementations may choose to offer $P_1..P_n$ as final destinations, allowing response packets to be forwarded back out on the network without host interaction. From each holding queue, packets are dequeued and (depending on the final destination) processed separately in the *DMA pipeline* or the *egress pipeline*. Both can again apply modifications.

If host memory is the final destination, the DMA pipeline issues requests to the DMA controller for transferring data between host memory and packet buffer. DMA parameters are passed to the DMA engine by adding a special DMA header to packets in the DMA pipeline, as shown in Table 1.

This design is easily extended to include support for CPU cache steering [20; 38] and atomic operations [37] if supported by the PCIe chipset. We can use this design to exchange packets with host memory or to carry out more complex memory exchanges. Packet checksums are calculated on the final result using the existing checksum offload capabilities.

Control flow. In general, actions execute in parallel, but FlexNIC also allows us to define dependencies among actions up to a hardware-defined limit. Compilers are responsible for rejecting code with too many dependencies for a particular hardware implementation. Dependencies are simply defined as "executes-before" relationships and may be chained. When mapping a rule set to a hardware pipeline, dependencies constrain which physical pipeline stages can be used for executing individual actions.

Field	Description
`offset`	Byte offset in the packet
`length`	Number of bytes to transfer
`direction`	From/To memory
`memBase`	Start address in memory
`cache`	Cache level to send data to
`core`	Core Id, if data should go to cache
`atomicOp`	PCIe atomic operation [37]

Table 1. Header format for DMA requests.

Such control flow dependencies are useful to restrict application-level access to packet filtering under kernel bypass [39; 5]. The OS inserts early ingress rules that assign incoming packets to a subset of the installed application rules by tagging packets with an identifier. Rule subsets are associated with applications by requiring them to match on the identifier. If such rules are inserted appropriately by the OS, then applications only receive packet flows deemed accessible by them (e.g., by matching on a specific MAC or IP address first).

Constraints. To make packet processing at high line-rates feasible, the RMT model is explicitly not freely programmable and several restrictions are imposed upon the user. For example, processing primitives are limited. Multiplication, division and floating point operations are typically not feasible. Hashing primitives, however, are available; this exposes the hardware that NICs use today for flow steering. Control flow mechanisms, such as loops and pointers, are also unavailable, and entries inside M+A tables cannot be updated on the data path. This precludes complex computations from being used on the data path. We expect the amount of stateful NIC memory to be constrained; in particular, our experimental results assume no per-flow state.

Programming language. The programming interface to FlexNIC is inspired by the P4 language [9] proposed for configuring programmable switches. Parsers, deparsers, packet headers, and M+A rules can be defined in much the same way, but we also allow handling NIC-internal SRAM in M+A stages, with additional actions for accessing this memory. Integration of the DMA controller, the other major change from RMT switches, does not require language extensions, but we add action verbs to simplify programming.

Example. Figure 3 shows the pseudo-code of a FlexNIC program that implements memory efficient scaling as described in the motivating example of Section 2.1. This program makes use of all the required elements of our model. To execute this program, the FlexNIC parser extracts the referenced packet fields out of incoming packets and passes them to the M+A stages in the ingress pipeline. RMT tables in the

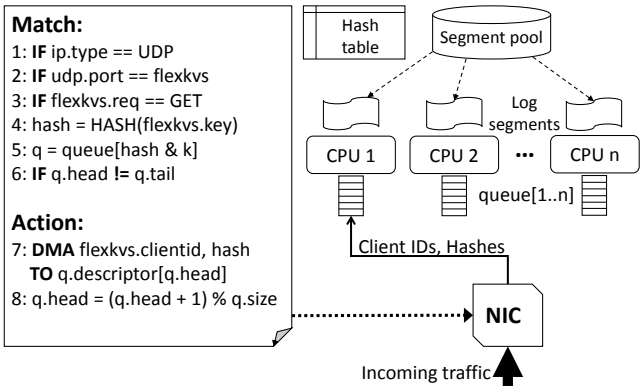

Figure 3. FlexNIC receive fast-path for FlexKVS: A rule matches GET requests for a particular key range and writes only the key hash together with a client identifier to a host descriptor queue. The queue tail is updated by FlexKVS via a register write.

ingress pipeline then determine whether the fields match the **IF** declarations in the program. The hash in line 4, which is needed to determine the destination queue, is computed for every incoming packet after parsing (for example, using the CRC mechanism used for checksums) and discarded if not needed later. Line 5 uses a match table to map ranges of hashes to queues of the cores responsible for those keys. `q.head` and `q.tail` in line 6 are maintained in the NIC-internal SRAM and are compared in the ingress pipeline. The action part is inserted into the DMA pipeline. The DMA pipeline generates packet headers to instruct the DMA engine to transfer the specified packet and metadata fields out of the pipeline buffers to the host memory address of the next queue entry. Finally, line 8 updates `q.head` in the NIC-internal SRAM in any pipeline stage of the DMA pipeline. A description of how the FlexKVS application integrates with the FlexNIC pipeline is deferred to Section 3.

2.3 Building Blocks

During the use case exploration of FlexNIC, a number of building blocks have crystallized, which we believe are broadly applicable beyond the use cases presented in this paper. These building blocks provide easy, configurable access to a particular functionality that FlexNIC is useful for. We present them in this section and will refer back to them in later sections.

Multiplexing: Multiplexing has proven valuable to accelerate the performance of our applications. On the receive path, the NIC has to be able to identify incoming packets based on arbitrary header fields, drop unneeded headers, and place packets into software-defined queues. On the send path, the NIC has to read packets from various application-defined packet queues, prepend the correct headers, and send them along a fixed number of connections. This building block is implemented using only ingress/egress M+A rules.

Flow and congestion control: Today's high-speed NICs either assume the application is trusted to implement congestion control, or, as in RDMA, enforce a specific model in hardware. Many protocols can be directly encoded in an RMT model with simple packet handling and minimal per-flow state. For example, for flow control, a standard pattern we use is to configure FlexNIC to automatically generate receiver-side acks using ingress M+A rules, in tandem with delivering the payload to a receive queue for application processing. If the application falls behind, the receive queue will fill, the ack is not generated, and the sender will stall.

For congestion control, enforcement needs to be in the kernel while packet processing is at user level. Many data centers configure their switches to mark an explicit congestion notification (ECN) bit in each packet to indicate imminent congestion. We configure FlexNIC to pull the ECN bits out before the packet stream reaches the application; we forward these to the host operating system on the sender to allow it to adjust its rate limits without needing to trust the application.

Other protocols, such as TCP, seem to be stateful and require complex per-packet logic; although we believe it possible to use FlexNIC to efficiently transmit and receive TCP packets, we leave that discussion for future work.

Hashing: Hashing is essential to the scalable operation of NICs today, and hashes can be just as useful when handling packets inside application software. However, they often need to be re-computed there, adding overhead. This overhead can be easily eliminated by relaying the hardware-computed hash to software. In FlexNIC, we allow flexible hashing on arbitrary packet fields and relay the hash in a software-defined packet header via ingress or DMA rules.

Filtering: In addition to multiplexing, filtering can eliminate software overheads that would otherwise be required to handle error cases, even if very few illegal packets arrive in practice. In FlexNIC, we can insert ingress M+A rules that drop unwanted packets or divert them to a separate descriptor queue for software processing.

2.4 Testbed Implementation

To quantify the benefits of FlexNIC, we leverage a combination of hardware and software techniques that we implement within a test cluster. Whenever possible, we re-use existing hardware functionality to achieve FlexNIC functionality. When this is impossible, we emulate the missing functionality in software on a number of dedicated processor cores. This limits the performance of the emulation to slower link speeds than would be possible with a hardware implementation of FlexNIC and thus favors the baseline in our comparison. We describe the hardware and software emulation in this section, starting with the baseline hardware used within our cluster.

Testbed cluster. Our evaluation cluster contains six machines consisting of 6-core Intel Xeon E5-2430 (Sandy

Bridge) systems at 2.2 GHz with 18MB total cache space. Unless otherwise mentioned, hyperthreading is enabled, yielding 12 hyperthreads per machine. All systems have an Intel X520 (82599-based) dual-port 10Gb Ethernet adapter with both ports connected to the same 10Gb Dell PowerConnect 8024F Ethernet switch. We run Ubuntu Linux 14.04.

Hardware features re-used. We make use of the X520's receive-side scaling (RSS) capabilities to carry out fast, customizable steering and hashing in hardware. RSS in commodity NICs operates on a small number of fixed packet header fields, such as IP and TCP/UDP source and destination addresses/ports, and are thus not customizable. We attain limited customization capability by configuring RSS to operate on IPv6 addresses, which yields the largest contiguous packet area to operate upon—32 bytes—and then moving the relevant data into these fields. This is sufficient for our experiments, as MAC addresses and port numbers are enough for routing within our simple network. RSS computes a 32-bit hash on the 32 byte field, which it also writes to the receive descriptor for software to read. It then uses the hash as an index into a 128-entry redirection table that determines the destination queue for an incoming packet.

Software implementation. We implement other needed functionality in a software NIC extension that we call Soft-FlexNIC. Soft-FlexNIC implements flexible demultiplexing, congestion control, and a customizable DMA interface to the host. To do this, Soft-FlexNIC uses dedicated host cores (2 send and 2 receive cores were sufficient to handle 10Gb/s) and a shared memory interface to system software that mimics the hardware interface. For performance, Soft-FlexNIC makes use of batching, pipelining, and lock-free queueing. Batching and pipelining work as described in [19]. Lock-free queueing is used to allow scalable access to the emulated host descriptor queue from multiple Soft-FlexNIC threads by atomically reserving queue positions via a compare and swap operation. We try to ensure that our emulation adequately approximates DMA interface overheads and NIC M+A parallelism, but our approach may be optimistic in modelling PCI round trips, as these cannot be emulated easily using CPU cache coherence interactions.

Baseline. In order to provide an adequate comparison, we also run all software on top of the high-performance Extaris user-level network stack [39] and make this our baseline. Extaris runs minimal code to handle packets and is a zero-copy and scalable network stack. This eliminates the overheads inherent in a kernel-level network stack and allows us to focus on the improvements that are due to FlexNIC.

3. Case Study: Key-Value Store

We now describe the design of a key-value store, FlexKVS, that is compatible with Memcached [2], but whose performance is optimized using the functionality provided by FlexNIC. To achieve performance close to the hardware limit, we needed to streamline the store's internal design, as we hit several scalability bottlenecks with Memcached. The authors of MICA [30] had similar problems, but unlike MICA, we assume no changes to the protocol or client software. We discuss the design principles in optimizing FlexKVS before outlining its individual components.

Minimize cache coherence traffic. FlexKVS achieves memory-efficient scaling by partitioning the handling of the key-space across multiple cores and using FlexNIC to steer incoming requests to the appropriate queue serving a given core. Key-based steering improves cache locality, minimizes synchronization, and improves cache utilization, by handling individual keys on designated cores without sharing in the common case. To support dynamic changes to the key assignment for load balancing, FlexKVS's data structures are locked; in between re-balancing (the common case), each lock will be cached exclusive to the core.

Specialized critical path. We further optimize FlexKVS by offloading request processing work to FlexNIC and specializing the various components on the critical path. FlexNIC checks headers, extracts the request payload, and performs network to host byte order tranformations. With these offloads, the FlexKVS main loop for receiving a request from the network, processing it and then sending a response consists of fewer than 3,000 x86 instructions including error handling. FlexKVS also makes full use of the zero-copy capabilities of Extaris. Only one copy is performed when storing items upon a SET request.
Figure 3 depicts the overall design, including a set of simplified FlexNIC rules to steer GET requests for a range of keys to a specific core.

3.1 FlexKVS Components

We now discuss the major components of FlexKVS: the hash table and the item allocator.

Hash table. FlexKVS uses a block chain hash table [30]. To avoid false sharing, each table entry has the size of a full cache line, which leaves room for a spin-lock, multiple item pointers (five on x86-64), and the corresponding hashes. Including the hashes on the table entries avoids dereferencing pointers and touching cache lines for non-matching items. If more items hash to an entry than there are available pointers the additional items are chained in a new entry via the last pointer. We use a power of two for the number of buckets in the table. This allows us to use the lowest k bits of the hash to choose a bucket, which is easy to implement in FlexNIC. k is chosen based on the demultiplexing queue table size loaded into the NIC (up to 128 entries in our prototype).

Item allocation. The item allocator in FlexKVS uses a log [44] for allocation as opposed to a slab allocator used in Memcached. This provides constant-time allocation and minimal cache access, improving overall request processing time. To minimize synchronization, the log is divided into

fixed-size segments. Each core has exactly one active segment that is used for satisfying allocation requests. A centralized segment pool is used to manage inactive segments, from which new segments are allocated when the active segment is full. Synchronization for pool access is required, but is infrequent enough to not cause noticeable overhead.

Item deletion. To handle item deletions, each log segment includes a counter of the number of bytes that have been freed in the segment. When an item's reference count drops to zero, the item becomes inactive and the corresponding segment header is looked up and the counter incremented. A background thread periodically scans segment headers for candidate segments to compact.When compacting, active items in the candidate segment are re-inserted into a new segment and inactive items deleted. After compaction, the segment is added to the free segment pool.

3.2 FlexNIC Implementation

The FlexNIC implementation consists of key-based steering and a custom DMA interface. We describe both.

Key-based steering. To implement key-based steering, we utilize the hashing and demultiplexing building blocks on the key field in the FlexKVS request packet, as shown in Figure 3. When enqueuing the packet to the appropriate receive queue based on the hash of the key, the NIC writes the hash into a special packet header for software to read. This hash value is used by FlexKVS for the hash table lookup.

Custom DMA interface. FlexNIC can also perform the item log append on SET requests, thereby enabling full zero-copy operation for both directions, and removal of packet parsing for SET requests in FlexKVS. To do so, FlexKVS registers a small number (four in our prototype) of log segments per core via a special message enqueued on a descriptor queue. These log segments are then filled by the NIC as it processes SET requests. When a segment fills up, FlexNIC enqueues a message to notify FlexKVS to register more segments.

FlexNIC still enqueues a message for each incoming request to the corresponding core, so remaining software processing can be done. For GET requests, this entails a hash table lookup. For SET requests, the hash table needs to be updated to point to the newly appended item.

Adapting FlexKVS to the custom DMA interface required adding 200 lines for interfacing with FlexNIC, adding 50 lines to the item allocator for managing NIC log segments, and modifications to request processing reducing the original 500 lines to 150.

3.3 Performance Evaluation

We evaluate different levels of integration between FlexKVS and the NIC. The baseline runs on the Extaris network stack using UDP and RSS, similar to the Memcached configuration in Arrakis [39]. A more integrated version uses FlexNIC to perform the hash calculation and steer requests based on their key, and the third version adds the FlexNIC custom DMA interface. FlexKVS performance is reduced by hyperthreading and so we disable it for these experiments, leaving 6 cores per machine.

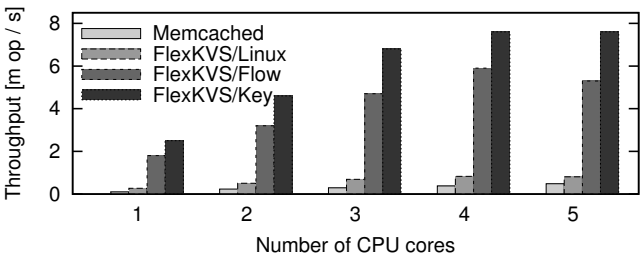

Figure 4. FlexKVS throughput scalability with flow-based and key-based steering. Results for Memcached and FlexKVS on Linux are also provided.

Key-based Steering. We compare FlexKVS throughput with flow-based steering against key-based steering using FlexNIC. We use three machines for this experiment, one server for running FlexKVS and two client machines to generate load. One NIC port of each client machine is used, and the server is connected with both ports using link aggregation, yielding a 20 Gb/s link. The workload consists of 100,000 key-value pairs of 32 byte keys and 64 byte values, with a skewed access distribution (zipf, $s = 0.9$). The workload contains 90% GET requests and 10% SET requests. Throughput was measured over 2 minutes after 1 minute of warm-up.

Figure 4 shows the average attained throughput for key- and flow-based steering over an increasing number of server cores. In the case of one core, the performance improvement in key-based steering is due to the computation of the hash function on the NIC. As the number of cores increases, lock contention and cache coherence traffic cause increasing overhead for flow-based steering. Key-based steering avoids these scalability bottlenecks and offers 30-45% better throughput. Note that the throughput for 4+ cores with key-based steering is limited by PCIe bus bandwidth, as the workload is issuing a large number of small bus transactions. We thus stop the experiment after 5 cores. Modifying the NIC driver to use batching increases throughput to 13 million operations per second.

Specialized DMA interface. The second experiment measures the server-side request processing latency. Three different configurations are compared: flow-based steering, key-based steering, and key-based steering with the specialized DMA interface described above. We measure time spent from the point an incoming packet is seen by the network stack to the point the corresponding response is inserted into the NIC descriptor queue.

Table 2 shows the median and 90th percentile of the number of cycles per request measured over 100 million requests.

	Steering		
	Flow	Key	DMA
Median	1110	690	440
90^{th} Percentile	1400	1070	680

Table 2. FlexKVS request processing time rounded to 10 cycles.

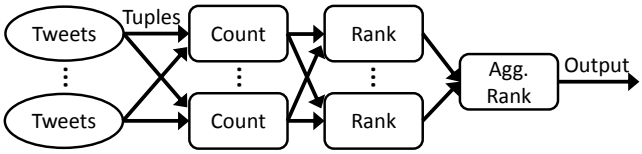

Figure 5. Top-n Twitter users topology.

Key-based steering reduces the number of CPU cycles by 38% over the baseline, matching the throughput results presented above. The custom DMA interface reduces this number by another 36%, leading to a cumulative reduction of 60%. These performance benefits can be attributed to three factors: 1) with FlexNIC limited protocol processing needs to be performed on packets, 2) receive buffer management is not required, and 3) log-appends for SET requests are executed by FlexNIC.

We conclude that flexible hashing and demultiplexing combined with the FlexNIC DMA engine yield considerable performance improvements in terms of latency and throughput for key-value stores. FlexNIC can efficiently carry out packet processing, buffer management, and log data structure management without additional work on server CPUs.

4. Case Study: Real-time Analytics

Real-time analytics platforms are useful tools to gain instantaneous, dynamic insight into vast datasets that change frequently. To be considered "real-time", the system must be able to produce answers within a short timespan (typically within a minute) and process millions of dataset changes per second. To do so, analytics platforms utilize data stream processing techniques: A set of *worker nodes* run continuously on a cluster of machines; data *tuples* containing updates stream through them according to a dataflow processing graph, known as a *topology*. Tuples are emitted and consumed worker-to-worker in the topology. Each worker can process and aggregate incoming tuples before emitting new tuples. Workers that emit tuples derived from an original data source are known as *spouts*.

In the example shown in Figure 5, consider processing a live feed of tweets to determine the current set of top-n tweeting users. First, tweets are injected as tuples into a set of counting workers to extract and then count the user name field within each tuple. The rest of the tuple is discarded. Counters are implemented with a sliding window. Periodically (every minute in our case), counters emit a tuple

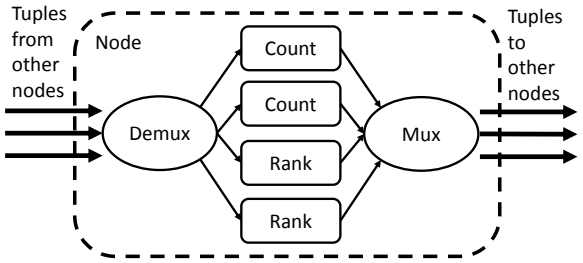

Figure 6. Storm worker design. 2 counters and 2 rankers run concurrently in this node. (De-)mux threads route incoming/outgoing tuples among network connections and workers.

for each active user name with its count. Ranking workers sort incoming tuples by count. They emit the top-n counted users to a single aggregating ranker, producing the final output rank to the user.

As shown in Figure 5, the system scales by replicating the counting and ranking workers and spreading incoming tuples over the replicas. This allows workers to process the data set in parallel. Tuples are flow controlled when sent among workers to minimize loss. Many implementations utilize the TCP protocol for this purpose.

We have implemented a real-time analytics platform FlexStorm, following the design of Apache Storm [49]. Storm and its successor Heron [28] are deployed at large-scale at Twitter. For high performance, we implement Storm's "at most once" tuple processing mode. In this mode, tuples are allowed to be dropped under overload, eliminating the need to track tuples through the topology. For efficiency, Storm and Heron make use of multicore machines and deploy multiple workers per machine. We replicate this behavior.

FlexStorm uses DCCP [27] for flow control. DCCP supports various congestion control mechanisms, but, unlike TCP, is a packet-oriented protocol. This simplifies implementation in FlexNIC. We use TCP's flow-control mechanism within DCCP, similar to the proposal in [17], but using TCP's cumulative acknowledgements instead of acknowledgement vectors and no congestion control of acknowledgements.

As topologies are often densely interconnected, both systems reduce the number of required network connections from per-worker to per-machine connections. On each machine, a demultiplexer thread is introduced that receives all incoming tuples and forwards them to the correct executor for processing. Similarly, outgoing tuples are first relayed to a multiplexer thread that batches tuples before sending them onto their destination connections for better performance. Figure 6 shows this setup, which we replicate in FlexStorm.

4.1 FlexNIC Implementation

As we will see later, software demultiplexing has high overhead and quickly becomes a bottleneck. We can use

Match:	Action:
IF ip.type == DCCP	SWAP(dccp.srcport, dccp.dstport)
IF dccp.dstport == FlexStorm	dccp.type = DCCP_ACK
	dccp.ack = dccp.seq
	dccp.checksum = CHECKSUM(dccp)
	IP_REPLY

Figure 7. Acknowledging incoming FlexStorm tuples in FlexNIC.

FlexNIC to mitigate this overhead by demultiplexing tuples in the NIC. Demultiplexing works in the same way as for FlexKVS, but does not require hashing. We strip incoming packets of their headers and deliver contained tuples to the appropriate worker's tuple queue via a lookup table that assigns destination worker identifiers to queues. However, our task is complicated by the fact that we have to enforce flow control.

To implement flow-control at the receiver in FlexNIC, we acknowledge every incoming tuple immediately and explicitly, by crafting an appropriate acknowledgement using the incoming tuple as a template. Figure 7 shows the required M+A pseudocode. To craft the acknowledgement, we swap source and destination port numbers, set the packet type appropriately, copy the incoming sequence number into the acknowledgement field, and compute the DCCP checksum. Finally, we send the reply IP packet, which does all the appropriate modifications to form an IP response, such as swapping Ethernet and IP source and destination addresses and computing the IP checksum.

To make use of FlexNIC we need to adapt FlexStorm to read from our custom queue format, which we optimize to minimize PCIe round-trips by marking whether a queue position is taken with a special field in each tuple's header. To do so, we replace FlexStorm's per-worker tuple queue implementation with one that supports this format, requiring a change of 100 lines of code to replace 4 functions and their data structures.

4.2 Evaluation

We evaluate the performance of Storm and various FlexStorm configurations on the top-n user topology. Our input workload is a stream of 476 million Twitter tweets collected between June–Dec 2009 [29]. Figure 8 and Table 3 show average achievable throughput and latency at peak load on this workload. Throughput is measured in tuples processed per second over a runtime of 20 seconds. Latency is also measured per tuple and is broken down into time spent in processing, and in input and output queues, as measured at user-level, within FlexStorm. For comparison, data center network packet delay is typically on the order of tens to hundreds of microseconds.

We first compare the performance of FlexStorm to that of Apache Storm running on Linux. We tune Apache Storm for high performance: we use "at most once" processing, disable all logging and debugging, and configure the optimum amount of worker threads (equal to the number of hyper-

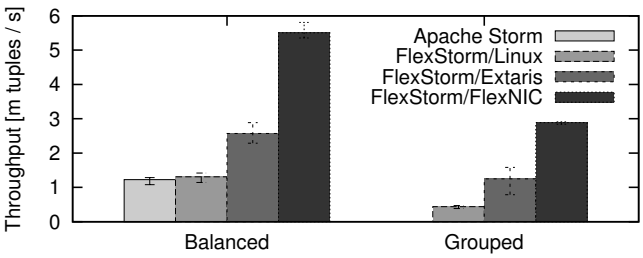

Figure 8. Average top-n tweeter throughput on various Storm configurations. Error bars show min/max over 20 runs.

threads minus two multiplexing threads). We deploy both systems in an identical fashion on 3 machines of our evaluation cluster. In this configuration Apache Storm decides to run 4 replicas each of spouts, counters and intermediate rankers. Storm distributes replicas evenly over the existing hyperthreads and we follow this distribution in FlexStorm (Balanced configuration). By spreading the workload evenly over all machines, the amount of tuples that need to be processed at each machine is reduced, relieving the demultiplexing thread somewhat. To show the maximum attainable benefit with FlexNIC, we also run FlexStorm configurations where all counting workers are executing on the same machine (Grouped configuration). The counting workers have to sustain the highest amount of tuples and thus exert the highest load on the system.

Without FlexNIC, the performance of Storm and FlexStorm are roughly equivalent. There is a slight improvement in FlexStorm due to its simplicity. Both systems are limited by Linux kernel network stack performance. Even though per-tuple processing time in FlexStorm is short, tuples spend several milliseconds in queues after reception and before emission. Queueing before emission is due to batching in the multiplexing thread, which is configured to batch up to 10 milliseconds of tuples before emission in FlexStorm (Apache Storm uses up to 500 milliseconds). Input queueing is minimal in FlexStorm as it is past the bottleneck of the Linux kernel and thus packets are queued at a lower rate than they are removed from the queue. FlexStorm performance is degraded to 34% when grouping all counting workers, as all tuples now go through a single bottleneck kernel network stack, as opposed to three.

Running all FlexStorm nodes on the Extaris network stack yields a 2× (Balanced) throughput improvement. Input queuing delay has increased as tuples are queued at a higher rate. The increase is offset by a decrease in output queueing delay, as packets can be sent faster due to the high-performance network stack. Overall, tuple processing latency has decreased 16% versus Linux. The grouped configuration attains a speedup of 2.84× versus the equivalent Linux configuration. The bottleneck in both cases is the demultiplexer thread.

	Input	Processing	Output	**Total**
Linux	6.68 μs	0.6 μs	12 ms	**12 ms**
Extaris	4 ms	0.8 μs	6 ms	**10 ms**
FlexNIC	–	0.8 μs	6 ms	**6 ms**

Table 3. Average FlexStorm tuple processing time.

Running all FlexStorm nodes on FlexNIC yields a 2.14× (Balanced) performance improvement versus the Extaris version. Using FlexNIC has eliminated the input queue and latency has decreased by 40% versus Extaris. The grouped configuration attains a speedup of 2.31×. We are now limited by the line-rate of our Ethernet network card.

We conclude that moving application-level packet demultiplexing functionality into the NIC yields performance benefits, while reducing the amount of time that tuples are held in queues. This provides the opportunity for tighter real-time processing guarantees under higher workloads using the same equipment. The additional requirement for receive-side flow control does not pose an implementation problem to FlexNIC. Finally, demultiplexing in the NIC is more efficient. It eliminates the need for additional demultiplexing threads, which can grow large under high line-rates.

We can use these results to predict what would happen at higher line-rates. The performance difference of roughly 2× between FlexNIC and a fast software implementation shows that we would require at least 2 fully utilized hyperthreads to perform demultiplexing for FlexStorm at a line-rate of 10Gb/s. As line-rate increases, this number increases proportionally, taking away threads that could otherwise be used to perform more analytics.

5. Case Study: Intrusion Detection

In this section we discuss how we leverage flexible packet steering to improve the throughput achieved by the Snort intrusion detection system [43]. Snort detects malicious activity, such as buffer overflow attacks, malware, and injection attacks, by scanning for suspicious patterns in individual packet flows.

Snort only uses a single thread for processing packets, but is commonly scaled up to multiple cores by running multiple Snort processes that receive packets from separate NIC hardware queues via RSS [19]. However, since many Snort patterns match only on subsets of the port space (for example, only on source or only on destination ports), we end up using these patterns on many cores, as RSS spreads connections by a 4-tuple of IP addresses and port numbers. Snort's working data structures generally grow to 10s of megabytes for production rule sets, leading to high cache pressure. Furthermore, the Toeplitz hash commonly used for RSS is not symmetric for the source and destination fields, meaning the two directions of a single flow can end up in different queues, which can be problematic for some stateful analyses where incoming and outgoing traffic is examined.

	Hashing			FlexNIC		
	Min	Avg	Max	Min	Avg	Max
Kpps	103.0	103.7	104.7	166.4	167.3	167.9
Mbps	435.9	439.8	444.6	710.4	715.6	718.6
Accesses	546.0	553.3	559.3	237.0	241.4	251.0
Misses	26.7	26.7	26.7	19.0	19.2	19.2

Table 4. Snort throughput and L3 cache behavior over 10 runs.

5.1 FlexNIC Implementation

Our approach to improve Snort's performance is similar to FlexKVS. We improve Snort's cache utilization by steering packets to cores based on expected pattern access, so as to avoid replicating the same state across caches. Internally, Snort groups rules into port groups and each group is then compiled to a deterministic finite automaton that implements pattern matching for the rules in the group. When processing a packet, Snort first determines the relevant port groups and then executes all associated automatons.

We instrument Snort to record each distinct set of port groups matched by each packet (which we call *flow groups*) and aggregate the number of packets that match this set and the total time spent processing these packets. In our experience, this approach results in 30-100 flow groups. We use these aggregates to generate a partition of flow groups to Snort processes, balancing the load of different flow groups using a simple greedy allocation algorithm that starts assigning the heaviest flow groups first.

This partitioning can then be used in FlexNIC to steer packets to individual Snort instances by creating a mapping table that maps packets to cores, similar to the approach in FlexStorm. We also remedy the issue with the Toeplitz hash by instructing FlexNIC to order 4-tuple fields arithmetically by increasing value before calculating the hash, which eliminates the assymetry.

5.2 Evaluation

We evaluate the FlexNIC-based packet steering by comparing it to basic hash-based steering. For our experiment, we use a 24GB pcap trace of the ICTF 2010 competition [1] that is replayed at 1Gbps. To obtain the flow group partition we use a prefix of 1/50th of the trace, and then use the rest to evaluate performance. For this experiment, we run 4 Snort instances on 4 cores on one socket of a two-socket 12-core Intel Xeon L5640 (Westmere) system at 2.2 GHz with 14MB total cache space. The other socket is used to replay the trace via Soft-FlexNIC implementing either configuration.

Table 4 shows the throughput and L3 cache behavior for hashing and our improved FlexNIC steering. Throughput, both in terms of packets and bytes processed, increases by roughly 60%, meaning more of the packets on the network are received and processed without being dropped. This improvement is due to the improved cache locality: the num-

ber of cache accesses to the L3 cache per packet is reduced by 56% because they hit in the L2 or L1 cache, and the number of misses in the L3 cache per packet is also reduced by roughly 28%. These performance improvements were achieved without modifying Snort.

We conclude that flexible demultiplexing can improve application performance and cache utilization even without requiring application modification. Application memory access patterns can be analyzed separately from applications and appropriate rules inserted into FlexNIC by the system administrator.

6. Other Applications

So far we discussed our approach in the context of single applications and provided case studies that quantify the application-level improvements made possible by FlexNIC. In this section, we identify additional use cases with some preliminary analysis on how FlexNIC can be of assistance in these scenarios. First, we discuss how to enhance and optimize OS support for virtualization (§6.1). Then, we present a number of additional use cases that focus on the use of flexible I/O to improve memory system performance (§6.2).

6.1 Virtualization Support

FlexNIC's functionality can aid virtualization in a number of different ways. We discuss below how FlexNIC can optimize the performance of a virtual switch, how it can provide better resource isolation guarantees, and how it can assist in transparent handling of resource exhaustion and failure scenarios.

Virtual Switch. Open vSwitch (OvS) [40] is an open source virtual switch designed for networking in virtualized data center environments. OvS forwards packets among physical and virtual NIC ports, potentially modifying packets on the way. It exposes standard management interfaces and allows advanced forwarding functions to be programmed via OpenFlow [31] M+A rules, a more limited precursor to P4 [9]. OvS is commonly used in data centers today. It supports various advanced switching features, including GRE tunneling.

To realize high performance packet forwarding without implementing all of OpenFlow in the OS kernel, OvS partitions its workflow into two components: a system userspace daemon and a simple kernel datapath module. The datapath module is able to process and forward packets according to simple 5-tuple and wildcard matches. If this is not enough to classify a flow, the packet is deferred to the userspace daemon, which implements the full matching capabilities of OpenFlow. After processing, the daemon relays its forwarding decision, potentially along with a concise 5-tuple match for future use, back to the datapath module.

To show the overhead involved in typical OvS operation, we conduct an experiment, sending a stream of packets from 4 source machines to 8 VMs executing on a single server machine that executes OvS to forward incoming packets to the appropriate VM. The traffic from the source machines is GRE tunneled and the tunnel is terminated at the server, so OvS decapsulates packets while forwarding. We repeat the experiment as we increase the offered traffic at the clients.

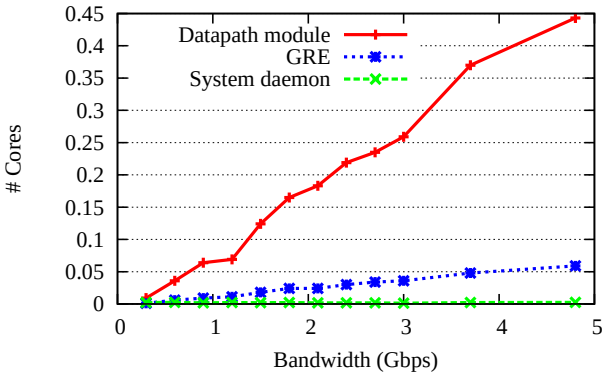

Figure 9. CPU utilization of OvS while tunneling.

Figure 9 shows the CPU overheads involved in this simple experiment, broken down into system daemon, datapath module, and GRE decapsulation overhead. We can see that the CPU overhead of the datapath module increases linearly with increasing traffic, as it has to carry out more matching and decapsulation activity. On average, the datapath module increases CPU utilization by 10% for each Gb of traffic offered. We can extrapolate these results to understand what would happen at higher line rates: We would require 1 full CPU core at 10 Gb/s, up to 4 CPU cores devoted to OvS handling on a 40 Gb/s link and up to 10 CPU cores on a 100 Gb/s link. The overhead for GRE decapsulation increases linearly, too, but at a slower pace: roughly 1% per offered Gb. The overhead of the system daemon stays flat, which is expected, as it is used only for more complex functionality.

To remove these overheads, we can implement the functionality of the kernel datapath module within FlexNIC. To do so, we simply offload the 5-tuple and wildcard match rules. If actions are as simple as GRE tunnel decapsulation, we can implement them in FlexNIC, too, by removing the relevant packet headers after matching, and gain even more savings. More complex actions might need to be relayed back to kernel-space, saving only the overhead of OvS processing.

We conclude that today's data center environments require flexible packet handling at the edge, which in turn requires CPU resources. While the amount of consumed CPU resource is tenable at 10 Gb/s, it can quickly become prohibitive as line rates increase. FlexNIC allows offloading a large fraction of these overheads, freeing up valuable CPU resources to run application workloads.

Resource isolation. The ability to isolate and prioritize network flows is important in a range of load conditions [3]. A server receiving more requests than it can handle could de-

cide to only accept specific request types, for example based on a priority specified in the request header. On servers that are not fully saturated, careful scheduling can reduce the average latency, for example by prioritizing cheap requests over expensive requests in a certain ratio and thereby reducing the overall tail latency. Achieving isolation implies minimizing the required resources that need to be invested before a priority can be assigned or a request can be dropped [14; 33]. As network bandwidth increases, the cost of classifying requests in software can quickly become prohibitive. FlexNIC can prioritize requests in hardware, and enqueue requests in different software queues based on priorities, or even reject requests, possibly after notifying the client.

Resource virtualization. Cloud services can be consolidated on a single server using resource virtualization. This allows the sharing of common hardware resources using techniques such as memory overcommit to save cost. However, traditional DMA operations require the corresponding host memory to be present, which is achieved by pinning the memory in the OS. With multi-queue NICs, kernel-bypass [39; 5], and RDMA, the amount of memory pinned permanently can be very large (cf. FaRM [13]), preventing effective server consolidation. Even without pinning, accesses to non-present pages are generally rare, so all that is required is a way to gracefully handle faults when they occur. Using FlexNIC, the OS can insert a rule that matches on DMA accesses to non-present regions and redirects them to a slow-path that implements these accesses in software. In this way these faults are handled in a manner fully transparent to applications.

Fast failover. User-level services can fail. In that case a hot standby or replica can be an excellent fail-over point if we can fail-over quickly. Using FlexNIC, an OS detecting a failed user-level application can insert a rule to redirect traffic to a replica, even if it resides on another server. Redirection can be implemented by a simple rule that matches the application's packets and then forwards incoming packets by rewriting the headers and enqueuing them to be sent out through the egress pipeline. No application-level modifications are necessary for this approach and redirection can occur with minimal overhead.

6.2 Flexible Memory Steering

FlexNIC's flexible packet processing provides additional opportunities for enhancing communication abstractions as well as optimizing memory system performance. We describe some of these use cases below.

Consistent RDMA. RDMA provides low latency access to remote memory. However, it does not support consistency of concurrently accessed data structures. This vastly complicates the design of applications sharing memory via RDMA (e.g., self-verifying data structures in Pilaf [32]). FlexNIC can be used to enhance RDMA with simple application-level consistency properties. For example, when concurrently writing to a hashtable, FlexNIC could check whether the target memory already contains an entry by atomically [37] testing and setting a designated memory location and, if so, either defer management of the operation to the CPU or fail it.

Cache flow control. A common problem when streaming data to caches (found in DDIO-based systems) is that the limited amount of available host cache memory can easily overflow with incoming network packets if software is not quick enough to process them. In this case, the cache spills older packets to DRAM causing cache thrashing and performance collapse when software pulls them back in the cache to process them. To prevent this, we can implement a simple credit-based flow control scheme: The NIC can increment an internal counter for each packet written to a host cache. If the counter is above a threshold, the NIC instead writes to DRAM. Software acknowledges finished packets by sending an application-defined packet via the NIC that decrements the counter. This ensures that packets stay in cache for as long as needed to keep performance stable.

GPU networking. General purpose computation capabilities on modern GPUs are increasingly being exploited to accelerate network applications [18; 24; 48; 26]. So far all these approaches require CPU involvement on the critical path for GPU-network communication. GPUnet [26] exploits P2P DMA to write packet payloads directly into GPU memory from the NIC, but the CPU is still required to forward packet notifications to/from the GPU using an in-memory ring buffer. Further, GPUnet relies on RDMA for offloading protocol processing to minimize inefficient sequential processing on the GPU. With FlexNIC, notifications about received packets can be written directly to the ring buffer because arbitrary memory writes can be crafted, thereby removing the CPU from the critical path for receiving packets. In addition FlexNIC's offload capabilities can enable the use of conventional protocols such as UDP, by offloading header verification to hardware.

7. Related Work

NIC-Software co-design. Previous attempts to improve the NIC processing performance used new software interfaces to reduce the number of required PCIe transitions [16] and to enable kernel-bypass [41; 50; 15]. Remote direct memory access (RDMA) goes a step further, entirely bypassing the remote CPU for this specific use-case. Scale-out NUMA [36] extends the RDMA approach by integrating a remote memory access controller with the processor cache hierarchy that automatically translates certain CPU memory accesses into remote memory operations. Portals [4] is similar to RDMA, but adds a set of offloadable memory and packet send operations triggered upon matching packet arrival. A machine model has been specified to allow offload

of various communication protocols using these operations [12]. Our approach builds upon these ideas by providing a NIC programming model that can leverage these features in an application-defined way to support efficient data delivery to the CPU, rather than complete offload to the NIC.

High-performance software design. High-performance key-value store implementations, such as HERD [25], Pilaf [32], and MICA [30], use NIC hardware features, such as RDMA, to gain performance. They require client modifications to be able to make use of these features. For example, they require clients to issue RDMA reads (Pilaf) or writes (HERD) to bypass network stacks, or for clients to compute hashes (Pilaf, MICA). FaRM [13] generalizes this approach to other distributed systems. In contrast, FlexNIC requires modifications only in the server's NIC.

Furthermore, FlexNIC allows logic to be implemented at the server side, reducing the risk of client-side attacks and allowing the use of server-sided state without long round-trip delays. For example, FlexKVS can locally change the runtime key partitioning among cores without informing the client—something that would otherwise be impractical with tens of thousands of clients.

MICA [30] can be run either in exclusive read exclusive write (EREW) or concurrent read exclusive write (CREW) mode. EREW eliminates any cache coherence traffic between cores, while CREW allows reads to popular keys to be handled by multiple cores at the cost of cache coherence traffic for writes and reduced cache utilization because the corresponding cache lines can be in multiple cores' private caches. FlexKVS can decide which mode to use for each individual key on the server side based on the keys' access patterns, allowing EREW to be used for the majority of keys and CREW to be enabled for very read heavy keys.

Programmable network hardware. In the wake of the software-defined networking trend, a rich set of customizable switch data planes have been proposed [9; 31]. For example, the P4 programming language proposal [9] allows users rich switching control based on arbitrary packet fields, independent of underlying switch hardware. We adapt this idea to provide similar functionality for the NIC-host interface. SoftNIC [19] is an attempt to customize NIC functionality by using dedicated CPU cores running software extensions in the data path. We extend upon this approach in Soft-FlexNIC.

On the host, various programmable NIC architectures have been developed. Fully programmable network processors are commercially available [35; 10], as are FPGA-based solutions [51]. Previous research on offloading functionality to these NICs has focused primarily on entire applications, such as key-value storage [7; 11] and map-reduce functionality [45]. In contrast, FlexNIC allows for the flexible offload of performance-relevant packet processing functionality, while allowing the programmer the flexibility to keep the rest of the application code on the CPU, improving programmer productivity and shortening software development cycles.

Direct cache access. Data Direct I/O (DDIO) [21] attaches PCIe directly to an L3 cache. DDIO is software-transparent and as such does not easily support the integration with higher levels of the cache hierarchy. Prior to DDIO, direct cache access (DCA) [20] supported cache prefetching via PCIe hints sent by devices and is now a PCI standard [38]. With FlexNIC support, DCA tags could be re-used by systems to fetch packets from L3 to L1 caches. While not yet supported by commodity servers, tighter cache integration has been shown to be beneficial in systems that integrate NICs with CPUs [6].

8. Conclusion

As multicore servers scale, the boundaries between NIC and switch are blurring: NICs need to route packets to appropriate cores, but also transform and filter them to reduce software processing and memory subsystem overheads. FlexNIC provides this functionality by extending the NIC DMA interface. Performance evaluation for a number of applications shows throughput improvements of up to 2.3× when packets are demultiplexed based on application-level information, and up to 60% reduction in request processing time if the DMA interface is specialized for the application on a commodity 10 Gb/s network. These results increase proportionally to the network line-rate.

FlexNIC also integrates nicely with current software-defined networking trends. Core network switches can tag packets with actions that can be executed by FlexNIC. For example, this allows us to exert fine-grained server control, such as load balancing to individual server cores, at a programmable core network switch when this is more efficient.

Acknowledgments

This work was supported by NetApp, Google, and the National Science Foundation (CNS-0963754, CNS-1518702). We would like to thank the anonymous reviewers and our shepherd, Steve Reinhardt, for their comments and feedback. We also thank Taesoo Kim, Luis Ceze, Jacob Nelson, Timothy Roscoe, and John Ousterhout for their input on early drafts of this paper. Jialin Li and Naveen Kr. Sharma collaborated on a class project providing the basis for FlexNIC.

References

[1] `http://ictf.cs.ucsb.edu/ictfdata/2010/dumps/`.

[2] `http://memcached.org/`.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 1999.

[4] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabee, and

T. Hudson. *The Portals 4.0.1 Network Programming Interface*. Sandia National Laboratories, sand2013-3181 edition, Apr. 2013.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[6] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2006.

[7] M. Blott, K. Karras, L. Liu, K. A. Vissers, J. Bär, and Z. István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *5th USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud, 2013.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *2013 ACM Conference on SIGCOMM*, SIGCOMM, 2013.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[10] Cavium Corporation. OCTEON II CN68XX multi-core MIPS64 processors. `http://www.cavium.com/pdfFiles/CN68XX_PB_Rev1.pdf`.

[11] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA Memcached appliance. In *21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, 2013.

[12] S. Di Girolamo, P. Jolivet, K. Underwood, and T. Hoefler. Exploiting offload enabled network interfaces. In *23rd IEEE Symposium on High Performance Interconnects*, HOTI, 2015.

[13] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[14] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 1996.

[15] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *1994 ACM Conference on SIGCOMM*, SIGCOMM, 1994.

[16] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *2013 USENIX Annual Technical Conference*, ATC, 2013.

[17] S. Floyd and E. Kohler. Profile for datagram congestion control protocol (DCCP) congestion control ID 2: TCP-like congestion control. RFC 4341, Mar. 2006.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated software router. In *2010 ACM Conference on SIGCOMM*, SIGCOMM, 2010.

[19] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html`.

[20] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[21] Intel Corporation. Intel data direct I/O technology (Intel DDIO): A primer, Feb. 2012. Revision 1.0. `http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf`.

[22] Intel Corporation. Flow APIs for hardware offloads. Open vSwitch Fall Confernce Talk, Nov. 2014. `http://openvswitch.org/support/ovscon2014/18/1430-hardware-based-packet-processing.pdf`.

[23] Intel Corporation. Intel 82599 10 GbE controller datasheet, Oct. 2015. Revision 3.2. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`.

[24] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2011.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *2014 ACM Conference on SIGCOMM*, SIGCOMM, 2014.

[26] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[27] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP). RFC 4340, Mar. 2006.

[28] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2015.

[29] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. `http://snap.stanford.edu/data`.

[30] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008.

[32] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference*, ATC, 2013.

[33] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[34] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *2014 Workshop on Memory Systems Performance and Correctness*, MSPC, 2014.

[35] Netronome. NFP-6xxx flow processor. `https://netronome.com/product/nfp-6xxx/`.

[36] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2014.

[37] PCI-SIG. Atomic operations. PCI-SIG Engineering Change Notice, Jan. 2008. `https://www.pcisig.com/specifications/pciexpress/specifications/ECN_Atomic_Ops_080417.pdf`.

[38] PCI-SIG. TLP processing hints. PCI-SIG Engineering Change Notice, Sept. 2008. `https://www.pcisig.com/specifications/pciexpress/specifications/ECN_TPH_11Sept08.pdf`.

[39] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[40] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2015.

[41] I. Pratt and K. Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. In *20th IEEE International Conference on Computer Communications*, INFOCOM, 2001.

[42] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. `http://www.rdmaconsortium.org/`.

[43] M. Roesch. Snort - lightweight intrusion detection for networks. In *13th USENIX Conference on System Administration*, LISA, 1999.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[45] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA. In *18th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, 2010.

[46] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We need to talk about NICs. In *14th Workshop on Hot Topics in Operating Systems*, HOTOS, 2013.

[47] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Googles datacenter network. In *2015 ACM Conference on SIGCOMM*, SIGCOMM, 2015.

[48] W. Sun and R. Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2013.

[49] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2014.

[50] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *15th ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[51] N. Zilberman, Y. Audzevich, G. Covington, and A. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept. 2014.