IncBricks: Toward In-Network Computation with an In-Network Cache

Ming Liu
University of Washington
mgliu@cs.washington.edu

Liang Luo
University of Washington
liangluo@cs.washington.edu

Jacob Nelson
Microsoft Research
jacob.nelson@microsoft.com

Luis Ceze
University of Washington
luisceze@cs.washington.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

Kishore Atreya
Cavium
kishore.atreya@cavium.com

Abstract

The emergence of programmable network devices and the increasing data traffic of datacenters motivate the idea of in-network computation. By offloading compute operations onto intermediate networking devices (e.g., switches, network accelerators, middleboxes), one can (1) serve network requests on the fly with low latency; (2) reduce datacenter traffic and mitigate network congestion; and (3) save energy by running servers in a low-power mode. However, since (1) existing switch technology doesn't provide general computing capabilities, and (2) commodity datacenter networks are complex (e.g., hierarchical fat-tree topologies, multipath communication), enabling in-network computation inside a datacenter is challenging.

In this paper, as a step towards in-network computing, we present IncBricks, an in-network caching fabric with basic computing primitives. IncBricks is a hardware-software co-designed system that supports caching in the network using a programmable network middlebox. As a key-value store accelerator, our prototype lowers request latency by over 30% and doubles throughput for 1024 byte values in a common cluster configuration. Our results demonstrate the effectiveness of in-network computing and that efficient datacenter network request processing is possible if we carefully split the computation across the different programmable computing elements in a datacenter, including programmable switches, network accelerators, and end hosts.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4465-4/17/04... $15.00

DOI: http://dx.doi.org/10.1145/3037697.3037731

Keywords in-network caching; programmable network devices

1. Introduction

Networking is a core part of today's datacenters. Modern applications such as big data analytics and distributed machine learning workloads [7, 11, 18, 31, 49] generate a tremendous amount of network traffic. Network congestion is a major cause of performance degradation, and many applications are sensitive to increases in packet latency and loss rates. Therefore, it is important to reduce traffic, lower communication latency, and reduce data communication overheads in commodity datacenters.

Existing networks move data but don't perform computation on transmitted data, since traditional network equipment (including NICs, switches, routers, and middleboxes) primarily focuses on achieving high throughput with limited forms of packet processing. Recently, driven by software-defined networking (SDN) and rapidly changing requirements in datacenter networks, new classes of programmable network devices such as programmable switches (e.g., Intel's FlexPipe [5], Cavium's XPliant [10], the Reconfigurable Match-Action Table architecture [16] and network accelerators (e.g., Cavium's OCTEON and LiquidIO products [9], Netronome's NFP-6000 [13], and FlexNIC [29] have emerged. Programmable switches allow for application-specific header parsing and customized match-action rules, providing terabit packet switching with a light-weight programmable forwarding plane. Network accelerators are equipped with scalable low-power multicore processors and fast traffic managers that support more substantial data plane computation at line rate. Together, they offer in-transit packet processing capabilities that can be used for application-level computation as data flows through the network.

The key idea of in-network computation (INC) is to offload a set of compute operations from end-servers onto programmable network devices (primarily switches and network accelerators) so that (1) remote operations can be served on the fly with low latency; (2) network traffic is reduced; and (3) servers can be put in low-power mode (e.g., Intel C6 state) or even be turned off or removed, leading to energy and cost savings.

However, offloading computation to the network has three challenges. First, even though the hardware resources associated with programmable network infrastructure have been improving over time, there is limited compute power and little storage to support general datacenter computation or services (like a key-value store). For instance, the Intel FlexPipe [5] chip in the Arista 7150S switch [6] has a flexible parser and a customizable match-action engine to make forwarding decisions and control flow state transitions. The switch is programmed with a set of rules, and then it applies data-driven modifications to packet headers as packets flow through the switch. The 9.5 MB packet buffer memory on this switch is not exposed for storing non-packet data; even if it was, the bulk of it would still be needed to buffer incoming traffic from dozens of ports in the case of congestion, leaving limited space for other uses. Network accelerators have less severe space and processing constraints, but that flexibility comes at the cost of reduced interface count and lower routing and traffic management performance when compared with switches. Second, datacenter networks offer many paths between end-points [24, 39, 44], and network component failures are commonplace, so keeping computations and state coherent across networking elements is complex. Finally, in-network computation needs a simple and general computing abstraction that can be easily integrated with application logic in order to support a broad class of datacenter applications.

In this paper we address the challenges outlined above and make the first step towards enabling in-network computation. We propose *IncBricks*, an in-network caching fabric with basic computing primitives, based on programmable network devices. IncBricks is a hardware/software co-designed system, comprising *IncBox* and *IncCache*. IncBox is a hybrid switch/network accelerator architecture that offers flexible support for offloading application-level operations. We choose a key-value store as a basic interface to the in-network caching fabric because it is general and broadly used by applications. We build IncCache, an in-network cache for key-value stores, and show that it significantly reduces request latency and increases throughput. IncCache borrows ideas from shared-memory multiprocessors and supports efficient coherent replication of key-value pairs. To support more general offloading, we provide basic computing primitives for IncCache, allowing applications to perform common compute operations on key-value pairs, such as increment, compare and update, etc. We prototype IncBox

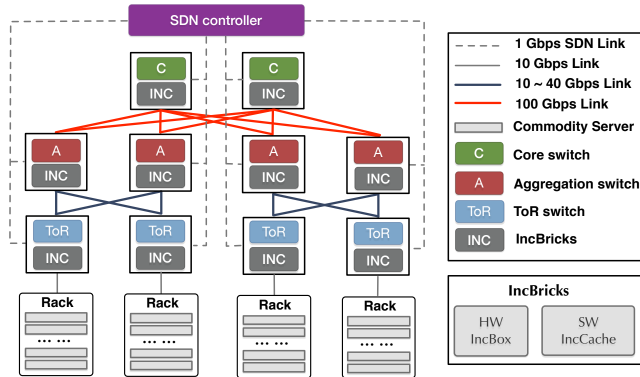


Figure 1. An overview of the IncBricks system architecture in a commodity datacenter network.

using Cavium's XPliant switches and LiquidIO boards. Our real system evaluations demonstrate that (1) in-network caching can provide common datacenter service abstractions with lower latency and higher throughput than existing software-only implementations; and (2) by carefully splitting the computation across programmable switches, network accelerators, and end hosts, one can achieve fast and efficient datacenter network request processing.

2. IncBricks system architecture

This section describes the IncBricks system architecture in the context of a datacenter. We first present the background of a commodity datacenter network and characteristics of programmable network switches and network accelerators, which are used in our work. Then we discuss the internal architecture of IncBricks in detail and explain how IncBricks integrates into existing networks.

2.1 Datacenter network

Figure 1 shows a typical datacenter network [24, 39, 45], with a hierarchical topology reaching from a layer of servers in racks at the bottom to a layer of core switches at the top. Each rack contains roughly 20 to 40 nodes, which connect to a Top of Rack (ToR) switch via 10 Gbps links. ToR switches connect to multiple aggregation switches (for both redundancy and performance) using 10–40 Gbps links, and these aggregation switches further connect to core switches with a higher bandwidth (e.g., 100 Gbps) link. To offer higher aggregate bandwidth and robustness, modern datacenters create multiple paths in the core of the network by adding redundant switches. Switches usually run a variant of ECMP [2] routing, hashing network flows across equal cost paths to balance load across the topology. Multiple paths bring coherence challenges to our IncBricks system design, which we discuss in Section 4.

Traditional Ethernet switches take an incoming packet and forward it based on a forwarding database (FDB). They comprise two parts: a data plane, which focuses on processing network packets at line rate, and a control plane, which

is used for configuring forwarding policies. The data plane consists of specialized logic for three core functions: (1) an ingress/egress controller, which maps transmitted and received packets between their wire-level representation and a unified, structured internal format; (2) packet memory, which buffers in-flight packets across all ingress ports; and (3) a switching module, which makes packet forwarding decisions based on the forwarding database. The control plane usually contains a low-power processor (e.g., an Intel Atom or ARM chip) that is primarily used for adding and removing forwarding rules. SDN techniques enable dynamic control and management by making the control plane more programmable [36, 43].

2.2 Programmable switch and network accelerator

Programmable switches add reconfigurability in their forwarding plane in order to overcome limitations of previous fixed-function switches [16]. Intel FlexPipe [5] and Cavium XPliant [10] are two examples. These switches include three core configurable units: a programmable parser, a match memory, and an action engine. This provides two benefits. First, packet formats are customizable and can be defined to suit application needs. For example, a key-value application can define a custom field that contains the key and the switch can perform match-action operations based on this field. Second, they support simple operations based on the headers of incoming packets. One can perform adaptive routing based on a custom field, store state on the switches based on packet contents, and perform packet transformations (e.g., header modifications) based on stored state.

Network accelerators provide fast packet processing with three major architectural components: (1) a traffic manager, supporting fast DMA between TX/RX ports and internal memory; (2) a packet scheduler, maintaining the incoming packet order and distributing packets to specific cores; (3) a low-power multicore (or manycore) processor, performing general payload modifications. Cavium's OCTEON/LiquidIO [9], Netronome's NFP-6000 [13], and Mellanox's NPS-400 [12] are recent commodity products that are widely used for deep packet inspection, SDN, and NFV applications. However, these accelerators support only a few interface ports, limiting their processing bandwidth.

In this work, we build a network middlebox by combining the above two hardware devices. We discuss our design decisions in Section 3.1 and show that one can achieve fast and efficient datacenter network request processing by taking advantage of all computing units along the packet path.

2.3 IncBricks

IncBricks is a hardware/software co-designed system that enables in-network caching with basic computing primitives. It comprises two components, IncBox and IncCache, shown in Figure 1.

IncBox is a hardware unit consisting of a network accelerator co-located with an Ethernet switch. After a packet marked for in-network computing arrives at the switch, the switch will forward it to its network accelerator, which performs the computation.

IncCache is a distributed, coherent key-value store augmented with some computing capabilities. It is responsible for packet parsing, hashtable lookup, command execution, and packet encapsulation.

Our IncBricks implementation highly depends on the execution characteristics of switches. For example, a core or aggregation switch with higher transmit rate and more ports might enclose a larger hash table inside its IncCache compared to the hash table of a ToR switch. Moreover, IncBricks requires that the datacenter has a centralized SDN controller connecting to all switches so that different IncBricks instances can see the same global view of the system state.

Next we explore the design and implementation of IncBricks' main components.

3. IncBox: Programmable Network Middlebox

This section describes IncBox, our middlebox. We first discuss our design decisions and then show the internal architecture of IncBox.

3.1 Design decisions

A design must support three things to enable in-network caching: (1) it must parse in-transit network packets and extract some fields for the IncBricks logic (**F1**); (2) it must modify both header and payload and forward the packet based on the hash of the key (**F2**); (3) it must cache key/value data and potentially execute basic operations on the cached value (**F3**). In terms of performance, it should provide higher throughput (**P1**) and lower latency (**P2**) than existing software-based systems for forwarding packets or for processing payloads.

A network switch is the ideal place to do in-network computation since all packets are processed by its forwarding pipeline. However, traditional fixed-function switches struggle to meet **F1** since their packet parser configuration is static. Software switches (such as Click [30], RouteBricks [22] and PacketShader [25]) are able to perform dynamic header parsing but they remain two orders of magnitude slower at switching than a hardware switch chip. This means they fail to meet **P1** and **P2** in terms of forwarding, not to mention header or payload modifications. Programmable switches seem a promising alternative. Their configurable parsers use a TCAM and action RAM to extract a customized header vector. During ingress processing, they could match on different parts of a packet based on user defined rules and apply header field modifications. Nevertheless, programmable switches have two drawbacks: first, they can support only simple operations, such as read or write to a

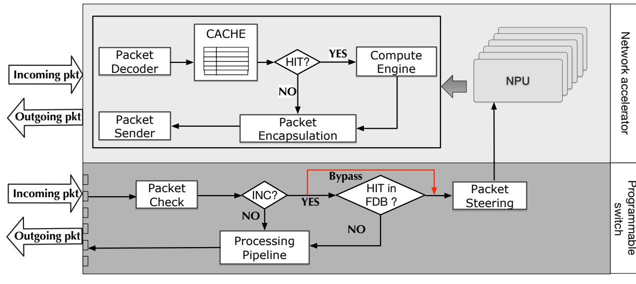


Figure 2. IncBox internal architecture.

specified memory address, or primitive compute operations (e.g., add, subtract, shift) on counters. There is no support for more complicated operations on payloads. Second, the size of packet buffer (along with TCAM and action RAM) is on the order of a few tens of megabytes; most of that is needed to store incoming packet traffic and match-action rules, leaving little space for caching. They meet **F1** and **F2**, but to satisfy **F3**, **P1**, and **P2** in terms of payload-related operations, we must take advantage of other devices.

We choose network accelerators to satisfy the rest of our requirements for three reasons. First, their traffic managers can serve packet data to a processing core in hundreds of nanoseconds (e.g., 200ns for our OCTEON board), which is significantly faster than kernel bypass techniques [41]. Second, their multicore processors are able to saturate 40Gbps–100Gbps of bandwidth easily, which is hard to achieve with general purpose CPUs. Third, they support multiple gigabytes of memory, which can be used for caching.

FPGA accelerators like Catapult [17] could also fit our scenario, but the general purpose processors in the network accelerator are more flexible and easier to program compared with a FPGA.

3.2 IncBox design and prototype

We carefully split IncBox's functionality across a programmable switch and a network accelerator. Figure 2 shows our design. The switch performs three tasks. The first is packet checking, which filters in-network caching packets based on the application header (Section 4.1). If there is a match, the packet will be forwarded to a network accelerator. Otherwise, it will be processed in the original processing pipeline. The second is key hit checks, which determines whether the network accelerator has cached the key or not. This is an optimization between the two hardware units (Section 4.5) and can be bypassed depending on the switch configuration. The third is packet steering, which forwards the packet to a specific port based on the hash value of the key. This will be used when there are multiple attached network accelerators, to ensure that the same key from the same network flow will always go the same accelerator, avoiding packet reordering issues. Since IncCache is based on the UDP protocol (Section 4.1), packet reordering will cause incorrect results (e.g., a GET after SET request might fail since it could be processed out of order). We expect that more capabilities will be provided in the upcoming programmable switches (i.e., Barefoot's Tofino) due to deeper ingress pipelines and more flexible rewrite engines. For example, a switch could maintain a Bloom filter of keys cached by the network accelerator or could even probabilistically identify heavy-hitter keys that are frequently accessed. Implementing these operations fully in the switch dataplane would require reading state updated by previous packets; the Cavium switch we use requires intervention of its management processor to make state updated by one packet available to another.

The network accelerator performs application-layer computations and runs our IncCache system. First, it extracts key/value pairs and the command from the packet payload. Next, it conducts memory related operations. If the command requires writing a new key/value pair, the accelerator will allocate space and save the data. If the command requires reading a value based on one key, the accelerator performs a cache lookup. If it misses, the processing stops and the network accelerator forwards the request further along its original path. If it hits, the network accelerator goes to the third stage and executes the command, both performing any operations on the value associated with the key as well as conducting cache coherence operations. Finally, after execution, the accelerator rebuilds the packet and sends it back to the switch.

Our IncBox prototype consists of (1) one 3.2 Tbps (32x100Gbps) Cavium XPliant switch and (2) one Cavium OCTEON-II many-core MIPS64 network accelerator card. We use two types of OCTEON-II devices: an OCTEON-II evaluation board with 32 CN68XX MIPS64 cores and 2GB memory, and a lightweight LiquidIO adapter, enclosing 8 CN66XX MIPS64 cores and 2GB memory. We program the switch by (1) adding packet parsing rules for the new packet header; (2) changing the ingress processing pipeline; and (3) modifying the forwarding database (FDB) handlers. We program the network accelerator using the Cavium Development kit (CDK), which provides a thin data plane OS (OCTEON simple executive) along with a set of C-language hardware acceleration libraries (such as compression/decompression, pattern matching, and encryption/decryption). Our software stack uses the hardware features of this network accelerator for fast packet processing.

4. IncCache: a distributed coherent key-value store

This section describes the IncCache system design. IncCache is able to (1) cache data on both IncBox units and end-servers; (2) keep the cache coherent using a directory-based cache coherence protocol; (3) handle scenarios related to multipath routing and failures; and (4) provide basic compute primitives.

Offset (UDP payload)	0	1	2	3
	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	Request ID		Magic Bytes	
4	Magic Bytes (cont.)			
8	Command			
12	Hash			
16 …	Payload (i.e. key, type, value length, value)			

Figure 3. In-network computing packet format

A common design pattern in datacenter applications today is to have two distributed storage layers: a high-latency persistent storage layer (like MySQL), and a low-latency in-memory cache layer (like Memcached). IncCache accelerates the latter; when data requests go through the datacenter network, our IncCache will (1) cache the value in the hash table (Section 4.2); (2) execute a directory-based cache coherence protocol (Sections 4.3, 4.4, 4.5); and (3) perform computation involving cached values (Section 4.6).

4.1 Packet format

Packet format is one of the key components enabling IncCache. There are three design requirements: (1) it should be a common format agreed to by the client, server, and switch; (2) it should be parsed by the network switch efficiently without maintaining network flow states; and (3) it should be flexible and easily extended.

We target UDP-based network protocols, which have been shown to reduce client-perceived latency and overhead [40] and are also amenable for switch processing without requiring per-flow state. Our in-network computing messages are carried in the UDP payload; their format is shown in Figure 3 and described in detail here:

- **Request ID (2 bytes):** A client-provided value for matching responses with their requests.
- **Magic field (6 bytes):** Labels in-network cache packets, allowing switches to filter other UDP packets that should not be handled by IncCache.
- **Command (4 bytes):** Selects which operation should be performed on the key and value in the payload.
- **Hash (4 bytes):** The hash of the key, which is used to guarantee ordering inside the network switch.
- **Application payload (variable bytes):** The key-value pair, including key, type, value length, and value.

4.2 Hash table based data cache

We use a hash table to cache the key-value data on both network accelerators and endhost servers. To achieve high throughput, the hash table should (1) support concurrent accesses from multiple processing cores and (2) provide full associativity for cached keys. It should also respect the constraints of the hardware.

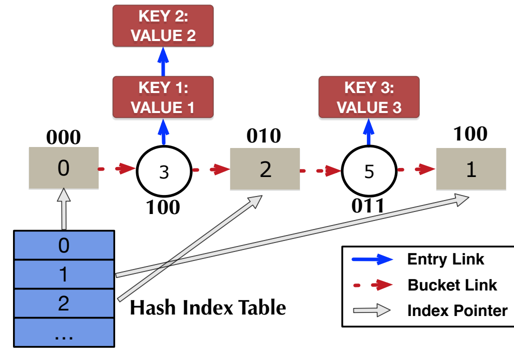


Figure 4. Bucket splitting hash table design. The prefix of a key's hash code is looked up in an index table, which points to (square) sentinel bucket nodes in the bucket list. Hash table entries are stored in entry lists from each bucket node. As the table expands, buckets are inserted to create (round) overflow bucket nodes and the index is extended, and more sentinel (square) nodes are added.

4.2.1 Fixed size lock-free hash table

We designed the hash table on the network accelerator to have fixed size due to the limited memory space. It consists of a set of buckets, each containing an ordered linked list of entries. Each node in the bucket list has a unique hash code and serves as the head of an entry list. Items with the same hash code but different keys are stored on the same entry list, in sorted order based on the key. The hash table uses lock-free operations to access and modify its state [26, 37]. For set or delete operations, the hash table interacts with the memory allocator module to allocate and free data. Our allocator maintains a fixed number of data items and applies a least frequently used eviction policy.

4.2.2 Extensible lock-free hash table

On the server side, we implemented an extensible hash table due to the server's more relaxed memory constraints. It is also lock-free, using the split-ordered list technique [47]. Figure 4 shows the structure of the hash table.

Search and insert are the two primary operations, and behave similarly. When looking for an item, the hash table first traverses the bucket list to match a hash code. If there is a match, the corresponding entry list is searched for an exact key match. In order to accelerate this lookup process, our hash table maintains a set of hash indexes that are shortcuts into the bucket list. We build the hash index by examining the prefix of the hash code of a given item. An insert operation follows a similar procedure, except that it atomically swaps in a new node (using compare-and-swap) if there is no match. To guarantee a constant time lookup, the hash table dynamically expands the size of the hash index and examines a progressively longer prefix of an item's hash code. The newer elements in the expanded hash index are ordered to appear after the elements that existed before the expansion,

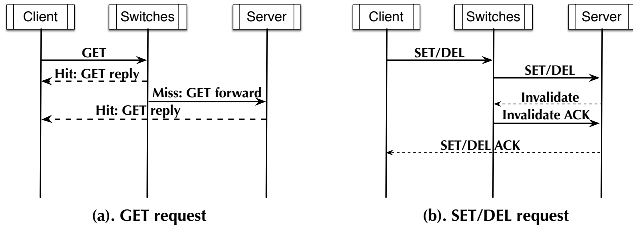


Figure 5. Packet flow for GET/SET/DELETE requests, showing how end-host servers and switches maintain cache coherence in a single path scenario.

resulting in what is known as a recursively split-ordered list. By recursively splitting the hash indexes with this technique, we introduce finer-grained shortcuts in a lock-free manner, without having to copy the index.

4.3 Hierarchical directory-based cache coherence

IncCache allows data to be cached in both IncBox units and servers. A cache coherence protocol is required to keep data consistent without incurring high overheads. Maintaining the sharers list (tracking who has the data) is a challenge since (1) servers don't know the details of the network topology, and (2) there is no shared communication media like the coherence bus in commodity processors.

Therefore, we propose a hierarchical directory-based cache coherence protocol. Our key ideas are: (1) take advantage of the structured network topology by using a hierarchical distributed directory mechanism to record the sharers information; (2) decouple the system interface and program interface in order to provide flexible programmability; (3) support sequential consistency for high performance SET/GET/DEL requests. We outline the scheme below, wherein we initially constrain the network topology to a tree topology and then generalize it to a multi-rooted multi-path topology in the subsequent section.

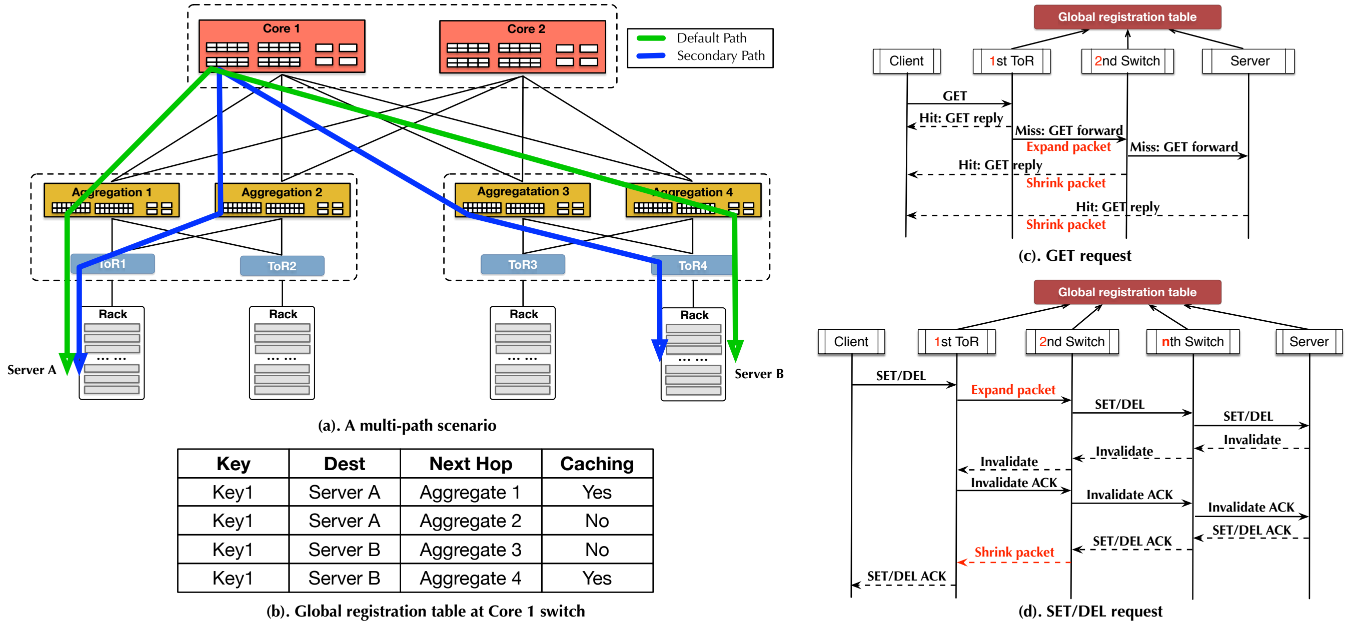
We place the home directory at the end-host server. Note that in the tree topology, a source-destination pair uniquely determines the entire communication path and all of the switches that can potentially cache the key-value entry. Hence, the directory only needs to record the source address to be able to infer all other potential sharers along the path. Switches (including ToR/Aggregation/Core) will route an "in-network" labeled packet to the IncBox unit for further processing. For a GET request (Figure 5-a), if there is a hit, the IncBox replies with the key's value directly. If not, the GET request will be forwarded to the end-host server to fetch the data. For a GET reply (Figure 5-a), each IncBox on the path will cache the data in the hash table. For SET and DELETE operations (Figure 5-b), we use a write invalidation mechanism. On a SET or DELETE request, switches first forward the request to the home node and then the home node issues an invalidation request to all sharers in the directory. IncBox units receiving this request will delete the data (specified in the request) from the hash table. The IncBox at the client-side ToR switch will respond with an invalidation ACK. After the home node receives all the expected invalidation ACKs, it performs the SET or DELETE operation and then sends a set/delete ACK to the client.

4.4 Extension for multi-rooted tree topologies

We discuss how to address the issue of multiple network paths between source-destination pairs as would be the case in multi-rooted network topologies. As existing data centers provide high aggregate bandwidth and robustness through multiple paths in a multi-rooted structure, this creates a challenge for our original cache coherence design—recording only the source/destination address is not sufficient to reconstruct the sharers list. For example, GET and SET requests might not traverse the same path, and a SET invalidation message might not invalidate the switches that were on the path of the GET response. We adapt our original cache coherence protocol to record path-level information for data requests and also traverse predictable paths in the normal case, thus minimizing the number of invalidate or update messages for the cache coherence module.

Figure 6-a shows a multi-path scenario. It has three layers of switches: ToR, Aggregation, and Core. Each switch connects to more than one upper layer switch for greater reliability and performance. We assume each switch routes based on ECMP. To handle the multi-path scenario, our key ideas are as follows. (1) ***Designated caching*** ensures that data is only cached at a set of predefined IncBox units for a given key. Any IncBox units between the designated IncBox units will not cache that key, meaning that routing between designated units can use any of the feasible paths. (2) ***Deterministic path selection*** ensures that for a given key and given destination, the end-host and IncBox units will choose a consistent set of designated caching IncBox units to traverse. (3) ***A global registration table*** (Figure 6-b), replicated on each IncBox, is used to identify the designated cache locations and designated paths for a given key and destination (indexed by a hash of the key rather than the key itself to bound its size). (4) ***A fault tolerance mechanism*** uses non-designated paths in the event of failures, but will not allow the data to be cached on the IncBox units traversed along the non-designated paths.

Our mechanism addresses the multi-path scenario by breaking a single (src, dest) flow into several shorter distance flows; for example, the path tuple (client, server) is transformed to a sequence of sub-paths: (client, designated ToR1) + (designated ToR1, designated Aggregation1/Core1) + ... + (designated ToR2, server). This means that we have to perform additional packet re-encapsulation operations at designated IncBox units along a designated path, which is not necessary in the single path case. The global registration table is preloaded on each IncBox and will be updated by a centralized SDN controller upon failures.



(a). A multi-path scenario

Key	Dest	Next Hop	Caching
Key1	Server A	Aggregate 1	Yes
Key1	Server A	Aggregate 2	No
Key1	Server B	Aggregate 3	No
Key1	Server B	Aggregate 4	Yes

(b). Global registration table at Core 1 switch

(c). GET request

(d). SET/DEL request

Figure 6. Cache coherence for a multi-rooted tree topology. (a) presents a typical multi-path scenario. Specially, it shows two communication paths between server A and server B; (b) gives an example about the global registration table and designated caching; (c) and (d) shows how to use proposed methods to maintain coherence for GET/SET/DELETE requests.

We illustrate our proposal using the example in Figure 6-a and Figure 6-b. The designated path between *Server A* and *Server B* for *Key1* is shown as the green/light colored path: *Server A* ↔ *ToR1* ↔ *Aggregation1* ↔ *Core1* ↔ *Aggregation4* ↔ *ToR4* ↔ *Server B*. Note that this is symmetric; communication in either direction uses the same path for a given key.

This designated path is always chosen for *Key1* when there are no failures. If *Core1* or *ToR1* detect a malfunction on one of *Aggregation4*'s links, they will instead use the secondary path through *Aggregation3* (the dark/blue path). However, given the global registration table at *Core1* (Figure 6-b), *Key1* will not be cached at *Aggregation3*. IncBox routes to the next designated caching location using network-layer ECMP, utilizing any of the paths connecting the IncBox to the next designated caching location, and skipping over the intermediate IncBox units.

We see two benefits of our design. First, the use of a designated path ensures that coherence messages traverse the same paths as the original cache request. It also increases the effectiveness of caching as the same key is not cached at multiple IncBox units at the same level of the network topology (other than ToR); this better utilizes total cache space and reduces invalidation overheads. Second, the ability to label designated caching locations provides a flexible policy mechanism that can take advantage of deployment and workload properties of the application. For example, Facebook deploys Memcached servers within a cluster and the clients in a different cluster. In such a setting, it might be more beneficial to perform caching only inside the client cluster as opposed to doing it also in the server cluster. The designated caching locations mechanism helps address such deployment concerns.

Now, we discuss in detail how to handle different requests:

GET requests and replies: As Figure 6-c shows, GET requests and replies behave similarly as in the single-path case. There are three differences. First, both IncBoxes and end servers calculate the next designated caching location based on the global registration table. Second, since the original packet flow has been transformed as discussed above, the IncBox unit at the first ToR switch (which connects to the client) will "expand" the packet to include a copy of the original client source address to be used in forming the reply. This address is removed from the packet (the packet is "shrunk") before sending the reply to the client. (3) For a GET reply, an IncBox will cache the data only if it is one of the designated caching locations.

SET/DEL requests and ACKs: As Figure 6-d shows, for a SET/DEL request, the IncBox unit will (1) interact with the global registration table and (2) perform packet expansion/shrink operations, which are the same as in GET requests and replies. Invalidation requests are generated by the home server. When a switch receives one, if the associated IncBox is not the destination, the packet will be forwarded; otherwise, it will be sent to its network processor, which will (1) invalidate the data, (2) calculate the next IncBox to be invalidated using the global registration table, and (3) modify the packet header and send it out. The IncBoxes at client-side ToRs respond to the home server with invalidation ACKs,

and the home server sends a SET/DEL ACK back to the requester when the coherence operation is complete (which is detected by counting the number of invalidation ACKs).

4.5 Optimization between switch and network processor

As described previously, the switch in an IncBox by default forwards any in-network caching requests to its network processor for further processing. For GET or DEL requests, if the key is not cached at the IncBox, this default operation will waste (1) one round trip between a switch and its network processor and (2) some processing time within the network processor. We observe that for a 1KB packet, these missed GET/DEL requests on average add 2.5us forwarding time at the switch and 2.7–5.0us computation time at the IncBox. This becomes even worse if it happens multiple times along one request path. Therefore, we propose an optimization that uses the switch's Forwarding Database (FDB) and associated management handlers to mitigate this overhead.

Our proposal works in the following way. First, we program the switch parser to extract the key hash field (Section 4.1) as well as MAC source and destination addresses. Second, we modify the behavior of the layer 2 processing stage of the switch's forwarding pipeline for certain in-network cache packets. For GET requests, instead of using the MAC destination address to perform FDB lookup, we use an artificial MAC address formed by concatenating a locally administered MAC prefix [4] with the key hash. If it's a hit, the packet is forwarded to the attached IncBox network processor that caches the key. Otherwise, the switch forwards it to the end server. For GET replies, after the IncBox network processor finishes cache operation execution and responds to the switch, we trigger the switch's MAC learning handler on its management CPU, using it to insert the key hash into the FDB. For invalidations, the ACK from the IncBox's network processor triggers another handler that removes the key hash from the FDB. To guarantee consistency, these switch FDB operations are triggered only after the IncBox network processor's work is complete.

This approach adds no additional latency to the switch's forwarding pipeline, and all non-INC packet types are processed exactly as before.

This approach has two limitations. First, two keys may have the same hash, so requests for keys not cached in an IncBox could be forwarded to its network processor. In this case the request is forwarded back towards the end server as it would be without this optimization. Two keys with the same hash may also be cached in the same IncBox; in this case, reference counts in the hash table can be used to ensure the FDB entry is not removed until both keys are evicted.

Second, the FDB has limited size due to the switch's resource constraints (16K entries in our testbed). To cache more than this many keys in an IncBox, a smaller, more collision-prone hash may be used in the FDB. Alternatively, heavy-hitter detection techniques may be used to identify a subset of frequently-accessed keys that can benefit from this optimization, while other keys would follow the normal, slower path.

4.6 Fault tolerance

In this section, we discuss how to handle failures. Generally, a failure must be one of the following: a client failure, a non designated switch/IncBox failure, a designated switch/IncBox failure, or a server failure. We categorize link failures as that of a switch/IncBox that is reached through that link.

Client failures could happen at three points in time: (1) before sending a request; (2) after sending a request but before receiving an ACK or data response packet; or (3) after receiving an ACK. None of these have an impact on the behavior of other clients and servers. Non-designated switch failure will result in path disconnection, in which case the preceding switch/IncBox would choose another path using ECMP. For a designated switch/IncBox failure, a request's whole communication path will be disconnected, and a backup path without caching will be employed until the global controller recomputes a new global registration table. Messages sent by a server (e.g., invalidations) could be lost, in which case the messages are retransmitted after a timeout. The server itself could fail either before or after sending the ACK or data result back. In the latter case, the data in the IncCache will be consistent but not available for updates. In the former case, the data will be inconsistent, cannot be updated, but the operation would not have been deemed complete, so the application can perform appropriate exception handling.

4.7 Compute primitives and client side APIs

IncCache's flexible packet format supports multiple compute primitives. While this work focuses primarily on the cache, we implemented two simple compute operations as a case study: conditional put and increment. They work as follows. When a IncBox or home server receives a compute request, it (1) reads the value from the cache; (2) if it is a miss, the request is forwarded; (3) if it is a hit, the IncBox or server directly performs the operation on the cached value (e.g., increments the value); (4) coherence operations are then performed; and (5) the response is sent back to the client. As should be apparent, there are many design tradeoffs about how the compute layer should interact with the caching layer. For example, when there is a key miss, it is also possible for the IncBox to issue a fetch request and perform the operation locally (as in a write-allocate cache). For this case study, we left our cache behavior unchanged. A more complete API and exploration of design tradeoffs is left to future work.

The client side API includes three classes of operations: (1) initialization/teardown calls, (2) IncCache accesses, and (3) compute case-study operations. `inc_init()` can be used to enable/disable the IncCache layer and configure the coherence protocol; `inc_close()` tears it down.

`inc_{get(), set(), del()}` read, write, and delete keys from the IncCache and block to guarantee data coherence. `inc_increment()` adds its integer value argument to a specified key. `inc_cond_put()` sets a key to its value argument if the cached value is smaller or unset.

4.8 IncCache Implementation

IncCache is implemented across end-host servers, switches, and network accelerators. The server side implementation is built following the model of Memcached and can support native Memcached traffic with modifications to the packet format. We do not take advantage of fast packet processing libraries (e.g., DPDK [1]) at present because most GET requests are served by IncBox units. Moreover, our server side could be easily replaced with a high performance key-value store, like MICA [33] or Masstree [35], with modifications to perform coherence operations.

We program the Cavium XPliant switch with its SDK, which includes C++ APIs to access hardware features from the management processor, as well as a proprietary language to program the forwarding pipeline.

The implementation of the network accelerator component is based on the Cavium Development Kit [8] and makes heavy use of the hardware features provided by the OCTEON-II network processor. We use a fixed length header for INC packets to accelerate parsing. For hashtable entries, we pre-allocate a big memory space (“arena” in the CDK) during the kernel loading phase and then use a private memory allocator (dlmalloc2) to allocate from that arena. There are multiple memory spaces in the processor, and we found that careful use is required to avoid considerable overhead. We also use the hardware-accelerated hash computation engine and the fetch and add accelerator (FAU) to hash keys and access lock-free data structures. As packet headers need frequent re-encapsulation, we use fast packet processing APIs to read and write the header. When generating a packet inside the accelerator, we take advantage of the DMA scatter and gather functions provided by the transmit units.

5. Evaluation

In this section, we demonstrate how IncBricks impacts a latency-critical datacenter application—an in-memory key-value store similar to Memcached [23]. Similar architectures are used as a building block for other datacenter workloads [18, 31]. We evaluate this with a real system prototype, answering the following questions:

- How does IncBricks impact the latency and throughput of a key-value store cluster under a typical workload setup?
- What factors will impact IncBricks performance?
- What are the benefits of doing computation in the cache?

Component	Description
Server	Dual-socket. A total of two Intel Xeon 6-core X5650 processors (running at 2.67GHz) and 24GB memory. The server is equipped with a Mellanox MT26448 10Gbit NIC.
Switch	Cavium/XPliant CN88XX-based switch with 32 100Gbit ports, configured to support 10Gbit links between emulated ToR switches and clients, and 40Gbit between emulated ToR and Root switches.
Network Accelerators	Two Cavium LiquidIO boards, each with one CN6640 processor (8 cnMIPS64 cores running at 0.8GHz) and 2GB memory. Two OCTEON-II EEB68 evaluation boards, with one CN6880 processor (32 cnMIPS64 cores running at 1.2GHz) and 2GB memory.

Table 1. Testbed details. Our IncBox comprises a Cavium switch and a network accelerator (OCTEON or LiquidIO).

5.1 Experimental setup

Platform. Our testbed is realized with a small-scale cluster that has 24 servers, 2 switches, and 4 network accelerators. Detailed information of each component is shown in Table 1. In our experiments, we set up a two-layer network topology where 4 emulated ToR switches connect to 2 emulated root switches; each ToR connects to both root switches. We emulate two ToR switches and one root switch in each Cavium switch by dividing the physical ports using VLANs. The IncBox at each ToR and Root switch is equipped with 4096 and 16384 entries, respectively, except for the cache size experiment in Section 5.3.1.

Workload. For the throughput-latency experiment, we set up a 4-server key-value store cluster and use the other 20 servers as clients to generate requests. We use our custom key-value request generator which is similar to YCSB [19]. Based on previous studies [15, 40], the workload is configured with (1) a Zipfian distribution [3] of skewness 0.99; (2) 95%/5% read/write ratio; (3) 1 million keys with 1024B value size; (4) two deployments: within-cluster and cross-cluster. The compute performance evaluation uses the full cluster with a uniform random load generator.

5.2 Throughput and latency

In this experiment, we compare throughput and latency of a vanilla key-value cluster with that of one augmented with IncBricks. Figures 7 and 8 report the average latency versus throughput under two cluster deployments. This comparison shows:

- At low to medium request rates (less than 1 million ops/s), IncBricks provides up to 10.4 us lower latency for the within-cluster case. The cross-cluster deployment provides an additional 8.6 us reduction because of the ad-

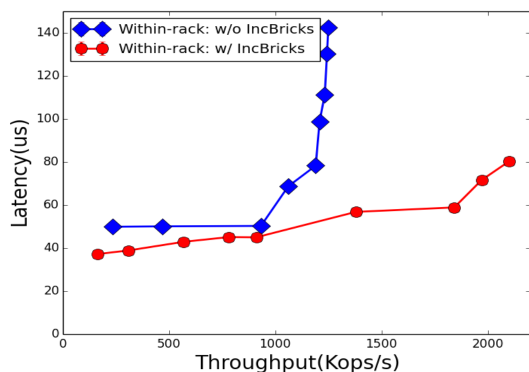


Figure 7. Average latency versus throughput in the within-cluster scenario

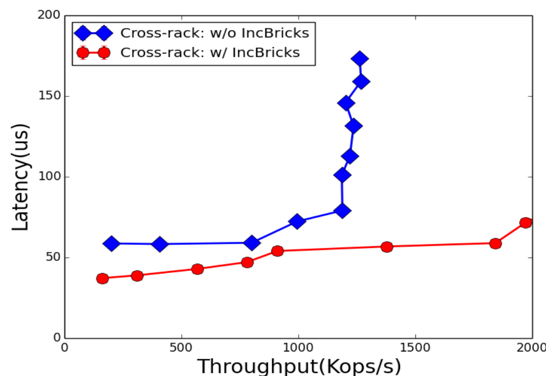


Figure 8. Average latency versus throughput in the cross-cluster scenario

ditional latency at Root and ToR switches. For high request rates (at 1.2 million ops/s), IncBricks saves even more: 85.8 us and 123.5 us for the within-cluster and cross-cluster cases, respectively.

- IncBricks is able to sustain a higher throughput than the vanilla key-value cluster for all scenarios. For example, IncBricks provides 857.0K ops/s and 458.4K ops/s more throughput for the within-cluster and cross-cluster scenarios, respectively. Furthermore, our measured maximum throughput is limited by the number of client nodes in our cluster and the rate at which they generate requests.
- This experiment does not include the optimization described in Section 4.5. If it did, our measurements suggest that we could save up to an additional 7.5 us in each IncBox, by reducing both (1) communication time between the switch and accelerator and (2) execution time inside the accelerator. This would provide significant benefit in the cross-cluster scenario since it has more hops.

5.2.1 SET request latency

IncBricks SET requests have higher latency than the vanilla case due to cache coherence overhead. In this section, we break down SET processing latency into four components: server processing, end-host OS network stack overhead, switch forwarding, and IncBox processing. Figure 9 shows this for both the within-cluster and cross-cluster scenarios.

First, server processing and the client and server OS network stack take the same amount of time (20 us) for both scenarios because the end-hosts behave the same in each. To reduce these components, we could apply kernel bypass techniques[1] or use a higher-performance key-value store instead (e.g., MICA[33] or Masstree[35]). Second, the switch forwarding latency increases from 21.5% of the total latency in the within-cluster case to 41.2% of the total latency in the cross-cluster case. This is due to the larger number of hops between clients and servers, which both requests and replies as well as invalidation traffic must traverse. Third, IncBox processing latency is higher (36 us) in the cross-cluster scenario compared with the within-cluster one (12 us) for the same reason. The optimization from Section 4.5 can help reduce this.

5.3 Performance factors of IncBricks

We explored two parameters that impact IncBricks performance: cache size and workload skewness.

5.3.1 Cache size

Figure 10 shows the result of varying the number of entries in the IncCache for both ToR and Root IncBoxes. We find that ToR cache size has a significant impact. As we increase the number of available cache entries from 512 to 16384, the average request latency decreases from 194.2 us to 48.9 us. Since we use a Zipfian distribution with skewness 0.99, 4096 entries covers 60% of the accesses, and 16384 entries covers 70%. That is why larger cache sizes provide less performance benefit. We also find that Root cache size has little impact on performance. This is due to our cache being inclusive; it is hard for requests that miss in the ToR IncBox to hit in a Root IncBox. This suggests that a ToR-only IncCache might provide the bulk of the benefit of the approach, or that moving to an exclusive cache model might be necessary to best exploit IncCaches at root switches.

5.3.2 Workload skewness

We vary the skewness from 0.99 to 0.01 and find that the average request latency at the client ToR IncBox increases significantly (at most 2 ms). Since our workload generates 1M keys and there are 4096 entries in the ToR IncCache, there will be many cache miss events as the workload access pattern becomes more uniform. These misses will fetch data from end-servers but also go through the IncBricks caching layer, adding latency. This may have two other consequences: the additional misses may cause network conges-

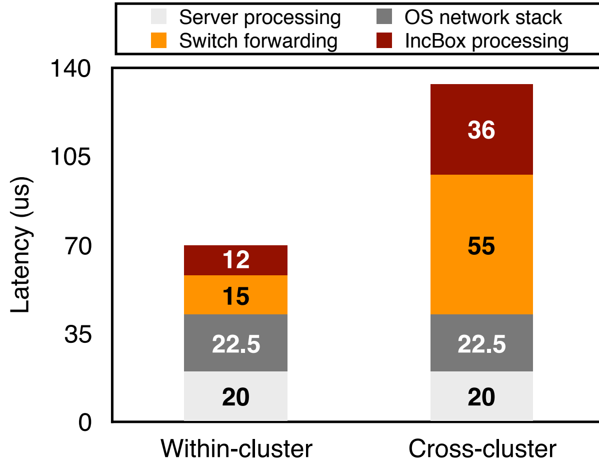


Figure 9. SET request latency breakdown for within-cluster and cross-cluster cases.

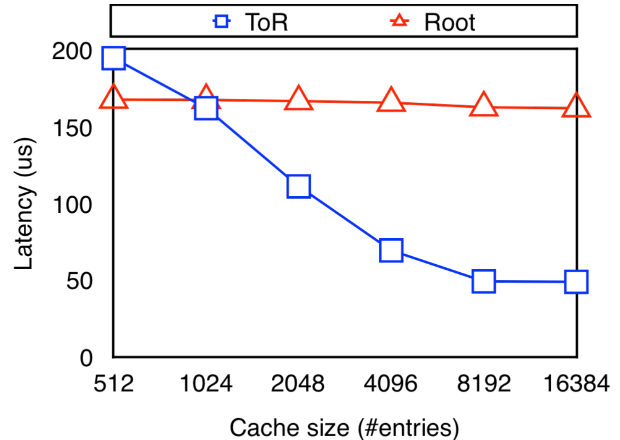


Figure 10. The impact of cache size at ToR and Root IncBoxes. y-axis is the average request latency.

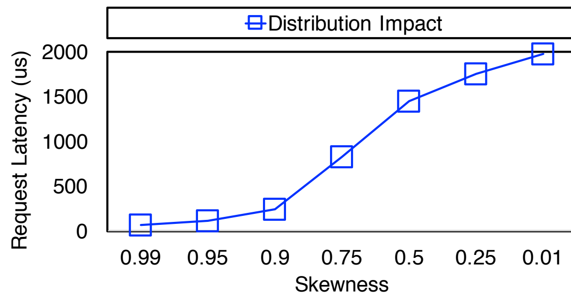


Figure 11. The impact of skewness on IncBricks for the cross-cluster case. y-axis is the average request latency.

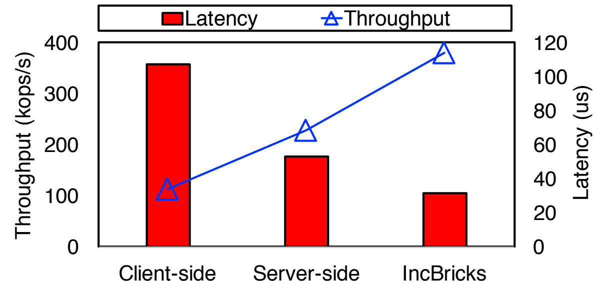


Figure 12. Performance comparison of the conditional put command among client-side, server-side, and IncBricks

tion on the key path, and the high miss rate may lead to high cache eviction overhead in the memory management layer of each IncBox unit. This suggests that a mechanism to detect "heavy-hitter" keys and bias the cache toward holding these hot keys could be beneficial. We plan to explore this in future work.

5.4 Case study: Conditional put and increment

This experiment demonstrates the benefit of performing compute operations in the network. As described in Section 4.7, we use conditional puts to demonstrate this capability. We compare three implementations: (1) client-side, where clients first fetch the key from the key-value servers, perform a computation, and then write the result back to the server, (2) server-side, where clients send requests to the server and the server performs the computation locally and replies, and (3) IncBricks, where computation is done in the network. We use a simple request generator for this experiment that generates uniform random operands that are applied to one of the keys, selected uniformly at random.

Figure 12 reports both latency and throughput for these three cases. The client-side implementation must perform both a GET and a SET for each operation, leading to 106.9 us latency and 113.3K ops/s. The server-side implementation only sends one message per remote operation and thus obtains better performance: 52.8 us and 228.1K ops/s. IncBricks performs the best since the operation can be performed directly by the IncBox, which results in 31.7 us latency and 379.5K ops/s. We also conducted the same experiment for increment operations, and the results were essentially identical and thus are omitted.

6. Related work

In-network aggregation. Motivated by the partition/aggregation model [14] used by many data center applications (e.g., [21, 27, 48]), in-network aggregation techniques have been developed to mitigate bandwidth constraints during the aggregation phase of these applications. Camdoop [20] exploits a low-radix directly-connected network topology to enable aggregation during the MapReduce shuffle phase. Servers are responsible for forwarding packets through the network, making computation on data traversing the network straightforward. However, this network is very differ-

ent from the high-radix indirect topologies commonly seen in datacenters today. NETAGG [34] targets similar aggregation opportunities, but on common datacenter network topologies, by attaching software middleboxes to network switches with high-capacity links. A set of shims and overlays route requests to middleboxes to be aggregated on their way to their final destination. In contrast to these largely-stateless approaches, IncBricks supports stateful applications with its coherent key-value cache. Data aggregation operations could be implemented as applications on top of IncBricks.

In-network monitoring. The limited compute capability found in switches today is largely intended for collecting network performance statistics (queue occupancy, port utilization, etc.) in order to support tasks such as congestion control, network measurement, troubleshooting, and verification. Minions [28] proposes embedding tiny processors in the switch data plane to run small, restricted programs embedded in packets to query and manipulate the state of the network. Smart Packets [46] explores a similar idea but in the switch control plane, with fewer restrictions on what could be executed, making it harder to sustain line rate. In contrast, the vision of IncBricks is to repurpose this in-switch compute capability (which was originally intended for these administrator-level tasks) and network accelerators, for user-level applications.

Network co-design for key-value stores. Dynamic load balancing is a key technique to ensure that scale-out storage systems meet their performance goals without considerable over-provisioning. SwitchKV [32] is a key-value store that uses OpenFlow-capable switch hardware to steer requests between high-performance cache nodes and resource-constrained backend storage nodes based on the content of the requests and the hotness of their keys. IncBricks also does content based routing of key-value requests, but by dispersing its cache throughout the network it can reduce communication in the core of the network, avoiding the hot spots SwitchKV seeks to mitigate.

7. Discussion

IncBricks is an in-network caching system, which we believe will be an important building block for a general in-network computing framework. The key observation in this work is that by carefully spreading computations among programmable switches, network accelerators, and end hosts, one can achieve fast and efficient datacenter network request processing. Toward this end, we plan to explore the following three topics in the future:

Programming model. As shown in this work, there are multiple computing devices along the path of a packet, each with their own unique hardware characteristics. It is difficult for programmers to fully utilize the aggregate computing power of all these devices without understanding their low-level hardware details. A good programming abstraction would not only reduce this burden but also provide a flexible interface for future networking hardware. In this direction, we plan to explore flow-based programming techniques [38] to design client-side APIs for more general workloads.

Runtime computation synthesizer (or scheduler). Efficiently mapping or scheduling computational tasks onto multiple network devices is a hard problem, since these computations have different execution characteristics, and the execution environment keeps changing. For example, a network accelerator might be overloaded under a read-intensive workload, and we might want to offload some requests to other networking devices or remote servers along the path. We plan to explore synthesis methods as in [42] to build an online scheduler.

Memory model for an in-network computing system. We'd like to provide a holistic in-network storage system, including both caching and persistent storage. It will likely be beneficial to provide multiple cache coherence protocols and consistency models for different storage mediums and application requirements. For example, a temporary local write might be handled more efficiently with a write-update coherence protocol and eventual consistency model, compared to our current approach.

8. Conclusion

This paper presents IncBricks, a hardware-software co-designed in-network caching fabric with basic computing primitives for datacenter networks. IncBricks comprises two components: (1) IncBox, a programmable middlebox combining a reconfigurable switch and a network accelerator, and (2) IncCache, a distributed key-value store built on IncBoxes. We prototype IncBricks in a real system using Cavium XPliant switches and OCTEON network accelerators. Our prototype lowers request latency by over 30% and doubles throughput for 1024 byte values in a common cluster configuration. When doing computations on cached values, IncBricks provides 3 times more throughput and a third of the latency of client-side computation. These results demonstrate the promise of in-network computation.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ada Gavrilvska, for their help and feedback. This work was supported in part by NSF under grants CCF-1518703, CSR-1518702, and CSR-1616774.

References

[1] Intel DPDK. http://dpdk.org.

[2] ECMP routing protocol. https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing.

[3] Zipf's law. https://en.wikipedia.org/wiki/Zipf%27s_law.

[4] Organizationally unique identifier. https://en.wikipedia.org/wiki/Organizationally_unique_identifier.

[5] Intel Ethernet Switch FM6000 Series, white paper, 2013.

[6] Arista 7150 Series Datasheet. https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf, 2016.

[7] Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning/, 2016.

[8] OCTEON Development Kits. http://www.cavium.com/octeon_software_develop_kit.html, 2016.

[9] LiquidIO Server Adapters. http://www.cavium.com/LiquidIO_Server_Adapters.html, 2016.

[10] XPliant Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html, 2016.

[11] Google SyntaxNet. https://research.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html, 2016.

[12] Mellanox NPS-400 Network Processor. http://www.mellanox.com/related-docs/prod_npu/PB_NPS-400.pdf, 2016.

[13] Netronome NFP-6000 Intelligent Ethernet Controller Family. https://www.netronome.com/media/redactor_files/PB_NFP-6000.pdf, 2016.

[14] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851192. URL http://doi.acm.org/10.1145/1851182.1851192.

[15] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1097-0. doi: 10.1145/2254756.2254766. URL http://doi.acm.org/10.1145/2254756.2254766.

[16] 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 *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486011. URL http://doi.acm.org/10.1145/2486001.2486011.

[17] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[18] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

[20] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228302.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[22] K. Fall, G. Iannaccone, M. Manesh, S. Ratnasamy, K. Argyraki, M. Dobrescu, and N. Egi. Routebricks: Enabling general purpose network infrastructure. *SIGOPS Oper. Syst. Rev.*, 45(1):112–125, Feb. 2011. ISSN 0163-5980. doi: 10.1145/1945023.1945037. URL http://doi.acm.org/10.1145/1945023.1945037.

[23] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1012889.1012894.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-594-9. doi: 10.1145/1592568.1592576. URL http://doi.acm.org/10.1145/1592568.1592576.

[25] 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. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851207. URL http://doi.acm.org/10.1145/1851182.1851207.

[26] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273005. URL http://doi.acm.org/10.1145/1272996.1273005.

[28] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceed-*

ings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, pages 3–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626292. URL http://doi.acm.org/10.1145/2619239.2626292.

[29] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with flexnic. *SIGPLAN Not.*, 51(4):67–81, Mar. 2016. ISSN 0362-1340. doi: 10.1145/2954679.2872367. URL http://doi.acm.org/10.1145/2954679.2872367.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[31] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[32] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, Mar. 2016. USENIX Association. ISBN 978-1-931971-29-4. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-xiaozhou.

[33] 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 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[34] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 249–262, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3279-8. doi: 10.1145/2674005.2674996. URL http://doi.acm.org/10.1145/2674005.2674996.

[35] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168855. URL http://doi.acm.org/10.1145/2168836.2168855.

[36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746. URL http://doi.acm.org/10.1145/1355734.1355746.

[37] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[38] J. P. Morrison. *Flow-Based Programming, 2Nd Edition: A New Approach to Application Development*. CreateSpace, Paramount, CA, 2010. ISBN 1451542321, 9781451542325.

[39] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-594-9. doi: 10.1145/1592568.1592575. URL http://doi.acm.org/10.1145/1592568.1592575.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[41] 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 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter.

[42] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594339. URL http://doi.acm.org/10.1145/2594291.2594339.

[43] L. Popa, N. Egi, S. Ratnasamy, and I. Stoica. Building extensible networks with rule-based forwarding. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 379–392, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924970.

[44] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0797-0. doi: 10.1145/2018436.2018467. URL http://doi.acm.org/10.1145/2018436.2018467.

[45] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787472. URL http://doi.acm.org/10.1145/2785956.2787472.

[46] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: Applying active networks to network management. *ACM Trans. Comput. Syst.*, 18(1):67–88, Feb. 2000. ISSN 0734-2071. doi: 10.1145/332799.332893. URL http://doi.acm.org/10.1145/332799.332893.

[47] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3), May 2006.

[48] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855742`.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863103.1863113`.