Key-value store cluster with programmability in NICs and switches CSE551 Final Report

Antoine Kaufmann (antoinek@), Naveen Kumar Sharma (naveenks@)

1 Introduction

Today's data center applications such as key-value stores generally run on multiple servers and on multiple cores for each sever in order to be able to handle the request loads. As a consequence, a steering and load-balancing decision needs to be made for each request about which server and core it should be handled by. Table 1 provides an overview of common approaches to load balancing, and how they are implemented for balancing across servers and across machines.

The different load-balancing options involve different trade-offs: When using dedicated load-balancer servers or cores, scaling up to high loads can be problematic. While using multiple severs for load-balancing is possible, one disadvantage is that the individual load-balancing servers no longer have a global view of the load which complicates dynamic load balancing. Dedicating cores in the end-servers for steering and balancing packets to worker cores can take up a significant number of cores for load balancing that cannot be used for processing requests. Implementing load balancing on the servers and server cores by forwarding requests uses up server resources for load balancing that could otherwise be used for processing requests. Further, forwarding packets also comes at the cost of increased latency and increased network/interconnect utilization. In the case where servers are forwarding requests to other servers, the servers also need up-to-date load information to forward requests to the right servers. A similar issue arises when the clients make load balancing decisions, because clients will also need up-to-date load information. This can significantly complicate dynamic load balancing.

Our project investigates using emerging programmable network hardware, namely switches [2] and NICs [4] to efficiently scale up a key-value store to multiple cores as well as multiple servers. Section 2 discusses how application-aware packet steering in the NIC improves scalability with increasing numbers of cores and describes the implementation of our scalable key-value store. In section 3 we discuss the advantages of implementing load-balancing inside the network in switches, and present a preliminary performance evaluation. Finally we discuss opportunities for further research in section 4.

2 Load Balancing across Cores

For a multi-core server running a key-value store requests need to be assigned to cores. To avoid the overhead of bouncing packet between cores, which can quickly become prohibitively expensive, we want the NIC to steer packets to the right core. Current NICs already support steering packets to multiple hardware queues that can be accessed from different cores without synchronization. Most commonly packets are assigned to hardware queues using receive-side scaling, in which case the NIC calculates a hash over packet fields for the IP addresses and port numbers and uses this hash to steer the packet to a hardware queue, which means that packets for a particular flow arrive on the same core. In the scenario of a key-value store a single connection generally issues a large number of requests and the popularity of keys is heavily skewed. This generally leads to synchronization and cache-coherence overheads for the indexing structure, most commonly a hash table, because popular items are accessed concurrently from multiple cores.

To avoid these overheads our goal is to use a more flexible NIC to steer requests to cores based on the key in the request. This effectively partitions accesses to the hash table, and therefor avoids synchronization and cache-coherence overheads. We then also went

Cores	Servers
Load-balancer core(s)	Load-balancer server(s)
Bounce requests to other cores	Forward requests to other servers
Client chooses port	Client chooses server

Table 1: Load-balancing techniques for balancing across cores and servers

one step further and investigated what other optimizations can be enabled by a programmable NIC by breaking with the traditional NIC DMA interface.

Note: The following section summarizes the results from Antoine's quals report, but in order to avoid duplicating the information there many details were omitted here, but are available in the quals report for reference.

2.1 Key-value store implementation

We implemented a simple key-value store from scratch with the goal of avoiding scalability bottleneck when requests arrive partitioned, and generally processing requests at high rates. The design uses an memory allocator for key-value pairs that is based on a segmented log per core and a centralized segment allocator. Thus, in the common case allocating memory just boils down to incrementing the log position. Garbage collection on the log is performed out of band on a separate core. The buffers storing key-value pairs are immutable to allow zero-copy operation without requiring synchronization. For indexing items we use a hash table where each bucket has the size of a cache line (to avoid false sharing), contains a lock and both pointers and the hashes of up to 5 elements. In the case of a bucket overflowing, the last element in the bucket is used as the head of a linked list of items.

2.2 Evaluation: key-based steering

Because we currently do not have access to programmable NICs, we emulate key-based steering with a regular Intel 10GbE NIC by having the client place the key for the request (we're assuming key-sizes ≤ 32 bytes) in the IPv6 source and destination address fields, and configure the NIC to only consider the IP-address for calculating the RSS hash. This allows us to implement key-based steering on the server side at line-rate, thereby enabling an end-to-end performance evaluation. In addition, the NIC also reports the calculated hash to software for every packet allowing us to use this hash for the hash-table lookup.

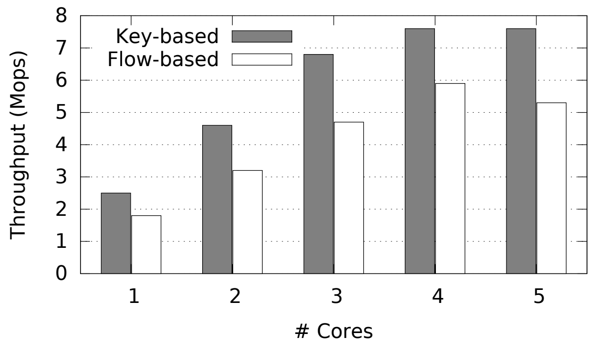

Figure 1: Key-value store throughput at varying numbers of cores comparing flow-based steering to key-based steering.

We compared the throughput of this RSS-based emulation against the baseline of regular connection-based RSS with varying numbers of cores. The experimental setup consists of a 6 core sandy-bridge machine with two aggregated 10GbE ports connected to a switch, and two client machines generating requests. The workload parameters are: 100K key-value pairs, 32 byte keys, 64 byte values, a Zipfian popularity distribution with parameter $s = 0.9$, and a mix of 90% GET + 10% SET operations. Figure 1 compares the throughput for both cases with different numbers of worker cores. As expected we see a significant throughput improvements (30-45%), and our results also hint at scalability improvements. Unfortunately performance is currently limited by PCIe throughput, because of the large number of small transactions. We expect that this can be somewhat alleviated by configuring the NIC to do more aggressive batching for PCIe transactions.

2.3 Evaluation: modified DMA interface

We also evaluated what performance gains are possible if the NIC provides a more flexible DMA interface, allowing us to customize the DMA interface

	Baseline	Steering	Opt. DMA
Median	1110	690	440
90th Percentile	1400	1070	680

Table 2: Request processing time in cycles for the baseline of flow-based steering, with key-based steering, and the specialized DMA interface.

to our key-value store use-case. With current NICs, packets are transferred to memory unmodified using descriptor queues. In order to receive packets, the application needs to first register a number of fixed-sized receive buffer, and when the NIC receives packets it will copy them in order into those buffers and hand ownership for the corresponding descriptors back to the application. Because the buffer sizes are fixed, they need to be sized for the worst case, and when the key-value store processes a SET request, it needs to copy the actual key-value pair from the packet into an appropriately sized item structure. All of this causes significant overhead when processing requests.

With a programmable NIC, we can have the NIC parse the packet and transfer the required information for software to process the request, based on the request type. For a GET request we simply append a client identifier, the hash, and the key to a contiguous event queue. In the case of a SET request, we append the client identifier and the hash to the event queue, and append the key-value pair to a separate item log. If we do this, the software part running on the CPU basically only manipulates the hash table and performs garbage collection on the log. We implemented an emulation of a NIC with this behavior in software, and ran it on a dedicated core. We then measured the number of cycles spent in software for processing a request, and compared it against the baseline with connection-based steering, as well as the key-based steering using RSS. The results are shown in Table 2, and show a cumulative reduction in request processing time of roughly 60%, of which 37% are due to the more efficient DMA interface.

3 Load Balancing across Servers

We take the approach in the previous section and apply it a layer above at the top-of-rack switch level. The flexibility inside the network switching chips allows us to route packets to desired servers based on the key embedded in the packet header and dynamic load metrics at the servers. The primary benefit of doing this is that the load-balancing can be done solely within the servers and network without involving the client, and achieving near perfect scalability. Moreover, the switches can update their forwarding decisions based on the dynamic load seen by any server or react faster in response to server failures.

This flexibility in switching hardware had been proposed by several manufacturers such as Intel, Cavium and Barefoot Networks. Most of them use a Reconfigurable Match-Action Tables (RMT) to parse and process custom packet headers. We can embed the key inside the packet header and use the switching chip to make forwarding decisions based on the key value. Moreover, we can populate the match-action tables with dynamic load information which can help the switch perform desired load balancing.

3.1 Evaluation: key-based routing

Since we don't have access to actual flexible switching chips, we evaluate key-based routing using the link aggregation feature in today's switches. Using link aggregation, multiple physical links are combined into a single logical channel and any packet is forwarded over exactly one physical link based on a hash of packet header fields. This provides load balancing over multiple links, as well as redundancy from failing links while maintaining connection affinity.

We emulate key-based routing on an Arista 7150 switch by having the client place the key for the request (assuming key-sizes 32 bytes) in the IPv6 source and destination address fields, and configure the switch to do link aggregation on outgoing server links using a hash over IP addresses and ports. This allows us to implement key-based routing on the switches at line-rate, thereby enabling an end-to-end performance evaluation.

TestBed Setup: We tested our implementation on our testbed consisting of 7 machines connected to the same Arista 7150 switch via 10Gbps links. We start off by a single server and increase the number of servers till 4 while link aggregating them at the switch. All other machines act as workload generating clients. Table 3 shows the increase in throughput we achieve by adding more servers. We obtain perfect scalability till 4 servers. The absolute throughput is low on purpose so that we can generate enough load to saturate the servers. This was done by adding an artificial delay while processing messages inside the key-value store. Further, we could not go beyond 4

# Servers	Throughput Ops/s
1	320,017 (1x)
2	633,712 (1.98x)
3	950,914 (2.97x)
4	1,296,183 (4.05x)

Table 3: Throughput with increasing servers

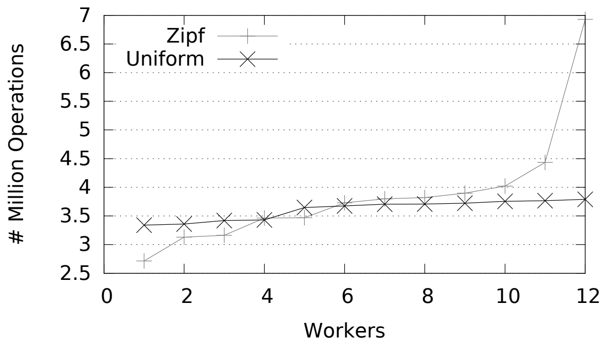

Figure 2: Resulting load-imbalance from static load balancing with Zipfian and uniformly distributed key popularity. Workers are sorted by increasing load.
.

servers due to lack of workload generating client machines connected to the same switch.

4 Future Work

This section discusses a number of options for future research on this project.

4.1 Dynamic load-balancing

So far we have implement simple hash-based static load balancing by calculating a hash over the key and then taking the hash modulo the number of cores to assign it to a core/server. Even assuming that key-popularity does not change over time, significant imbalances can occur, which require more advanced balancing techniques.

Figure 2 shows how requests are distributed to cores for the workload mentioned above, both with the Zipfian distribution for key popularity used above and also with a uniform distribution. For the uniform distribution the core with the largest number of requests received 13% more requests than the core with smallest number of requests. With the Zipfian distribution, the differences are significantly more pronounced, and the core with the heaviest load processed 155% more requests than the most lightly loaded core.

This can be addressed with dynamic load balancing, where requests are assigned to cores based on the current load instead of statically. In a class project for CSE521 (Algorithms) this quarter, Antoine started investigating an implementation of this using two algorithmic techniques: balanced allocation with power of 2 choices, and heavy hitters with the count-min sketch. The approach tries to avoid load imbalance by steering requests for popular keys to any of d candidate servers based on hashing, which significantly reduces load imbalance (there are theoretic results quantifying the expected benefits), without distributing the majority of less popular keys over multiple servers which would result in increased memory requirements and reduced locality. The count min-sketch used is a technique to probabilistically identify popular keys with sub-linear space requirements on the load balancer. Preliminary evaluation has shown that these benefits hold in practice, but a full performance evaluation has not been performed yet. Further, we expect that this technique could be implemented on the data plane of a programmable switch in the abstractions provided by P4 [1], a high-level programming language for programmable network hardware.

4.2 Performance tuning and additional evaluation

We are confident that further optimizations can be implemented for increasing the throughput that can be achieved by our key-value store. One example of this is making more efficient use of the PCIe bus by configuring the NIC to batch descriptor write-backs etc. There are also a number of other parameter that can be tuned on the NIC. In addition we also expect that significant throughput gains are enabled by employing techniques to hide latency for memory accesses by using batching and manual prefetching [3].

We are also interested in extending the software simulation for the flexible DMA interface to be able to use multiple cores for simulating the NIC. This would allow us to also get some throughput benchmarks. In addition the software emulation could also be used to get performance estimations for an architecture where the NIC is more closely integrated with the CPU, thereby getting rid of PCIe overheads.

4.3 Other applications

Finally, we are planning to evaluate the benefits of a programmable NIC and programmability inside the network for other applications. We believe that similar techniques can be applied to other request-response applications. But other applications such as packet inspection could also benefit from programmable network hardware.

References

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[2] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *2013 ACM Conference on SIGCOMM*, SIGCOMM, 2013.

[3] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using gpus in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2015.

[4] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. FlexNIC: Rethinking network DMA. In *15th Workshop on Hot Topics in Operating Systems*, HOTOS, 2015.