Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores

Diego Didona, *EPFL;* Willy Zwaenepoel, *EPFL and University of Sydney*

https://www.usenix.org/conference/nsdi19/presentation/didona

This paper is included in the Proceedings of the
16th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores

Diego Didona
EPFL

Willy Zwaenepoel
EPFL and University of Sydney

Abstract

This paper introduces the concept of size-aware sharding to improve tail latencies for in-memory key-value stores, and describes its implementation in the Minos key-value store.

Size-aware sharding distributes requests for keys to cores according to the size of the item associated with the key. In particular, requests for small and large items are sent to disjoint subsets of cores. Size-aware sharding improves tail latencies by avoiding that a request for a small item gets queued behind a request for a large item.

Minos uses hardware dispatch for all requests for small items, which form the very large majority of all requests, to achieve high throughput, and achieves load balancing by adapting the number of cores handling requests for small and large items to their relative presence in the workload.

We compare Minos to three state-of-the-art designs of in-memory KV stores. Compared to its closest competitor, Minos achieves a 99th percentile latency that is up to 20 times lower. Put differently, for a target 99th percentile latency equal to 10 times the mean service time, Minos achieves a throughput that is up to 7.4 times higher.

1 Introduction

Many distributed applications use in-memory key-value (KV) stores as caches or as (non-persistent) data repositories [3, 10, 13, 34, 41, 44, 51, 55]. Many of these applications exhibit a high fan-out pattern, i.e., they issue a large number of requests in parallel [55]. From the application's standpoint, the overall response time is then determined by the slowest of the responses to these requests, hence the crucial importance of tail latency for KV stores [17].

The performance of KV stores has been the subject of much work, both in terms of software and hardware. Software optimizations include zero-copy network stacks, polling, run-to-completion processing, and sharding of requests among cores [37, 45, 57]. Hardware optimizations primarily rely on the use of RDMA [35, 36], programmable NICs [38, 41] or GPUs [30, 64]. The work reported in this paper does not require any particular hardware support. We assume only commodity NICs with multiple queues and a hardware mechanism to direct requests to a particular queue.

Variable item sizes and tail latency. The workload observed for many KV stores consists of a very large number of requests for small items and a much smaller number of requests for large items [3, 9, 55]. Because of their higher service times, however, handling the requests for larger items consumes a significant share of the available resources. Processing these large items therefore increases the probability of head-of-line blocking, a situation in which a request for a small item ends up waiting while a large item is being processed. As a result of the wait, that request experiences additional latency, which in turn may increase the tail latency of the KV store. Even a very small number of requests for large items can significantly drive up tail latencies. As we show in Section 2.2, a percentage of large requests smaller than N% can lead to a substantial increase of the (100-N)th percentile.

Size-aware sharding. This paper introduces the notion of size-aware sharding to address this issue. In general, size-aware sharding means that requests for items of different sizes go to different cores. In its simplest form, it means that, for some cutoff value between small and large, small and large items are served by disjoint sets of cores. The intuition behind size-aware sharding is that by isolating the requests for small items, they do not experience any head-of-line blocking, and, given that they account for a very large percentage of requests, the corresponding percentile of the latency distribution is improved.

The implementation of size-aware sharding poses several challenges. A first challenge is how to use hardware dispatch of an incoming request to the right core. In general, a client of the KV store does not know the size of an item to be read, and moreover it does not know which cores are responsible for small or large items. Therefore, size-aware sharding would seem to necessitate a software handoff in which an I/O core reads incoming requests and dispatches them to the proper core. Instead, we demonstrate a method by which software dispatch is required only for the very small number of requests for large items. Second, cutoff values between large and small items must be chosen and the proper number of cores must be allocated for handling small and large items. We show that, even in the presence of a workload that varies over time, this can be done by a simple control loop.

Minos. This paper describes the Minos in-memory KV store that implements size-aware sharding. We compare Minos to alternative size-unaware designs based on keyhash-based request sharding, software handoff and work stealing, implemented by state-of-the-art systems such as MICA [45], RAMCloud [57] and ZygOS [58].

We show that Minos achieves a 99th percentile latency that is up to two 20 times lower than the second best approach. Put differently, for a given value for the 99th percentile latency equal to 10 times the mean service time, Minos achieves a throughput that is up to 7.4 times higher.

Contributions. The contributions of this paper are:

1) the introduction of the notion of size-aware sharding for in-memory KV stores,

2) the design and implementation of the Minos KV store that implements size-aware sharding efficiently, and

3) the evaluation of Minos against state-of-the-art size-unaware designs.

Outline of the paper. Section 2 provides background on KV store workloads and discusses the shortcomings of existing approaches in achieving low tail latency. Section 3 presents Minos' size-aware sharding approach. Section 4 discusses implementation details. Section 5 describes the experimental environment. Section 6 presents experimental results. Section 7 discusses related work. Section 8 concludes the paper.

2 Background

2.1 Item Sizes in Production KV Workloads

The sizes of the items stored and manipulated by KV stores in production environments can span orders of magnitude. For instance, large variations in item size have been reported in several deployments of the popular `memcached` KV store [51]. The Facebook `ETC` `memcached` pool stores items that vary in size from a handful of bytes to 1 Mbyte [3]. The size distribution is heavy-tailed: the 5th percentile in the `regional` pool is 231 bytes, while the 99th percentile is 381KB [55]. A similar degree of variability in item size has also been reported for other KV deployments such as Wikipedia [46] and Flickr [9], where item sizes span up to 4 orders of magnitude, from 500B to 1 MB.

Moreover, Atikoglu et al. report that in the `ETC` `memcached` pool at Facebook requests for large items, despite being rare, consume a large share of the computational resources, because service times are closely related to item size, and account for a significant fraction of the transfered data [3]. This dynamic is consistent with observations from similar application domains, such as, e.g., web servers [2, 15] and large-scale clusters [62].

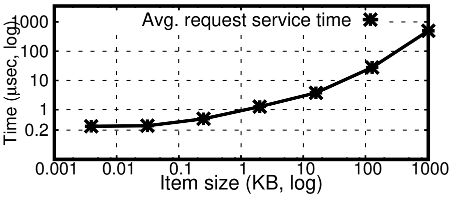


Figure 1: Service time of GET operations on items of different sizes on our platform (axes in log scale). The service time measures the interval from the reception of the client request on the server to the transmission of the reply. To avoid queueing effects, only one client performs operations. The time to process a large item can be up to almost four orders of magnitude higher than what is needed for a small one. This is due to the higher time needed to copy the content of the item to the network packets that are placed on the TX queue of the NIC.

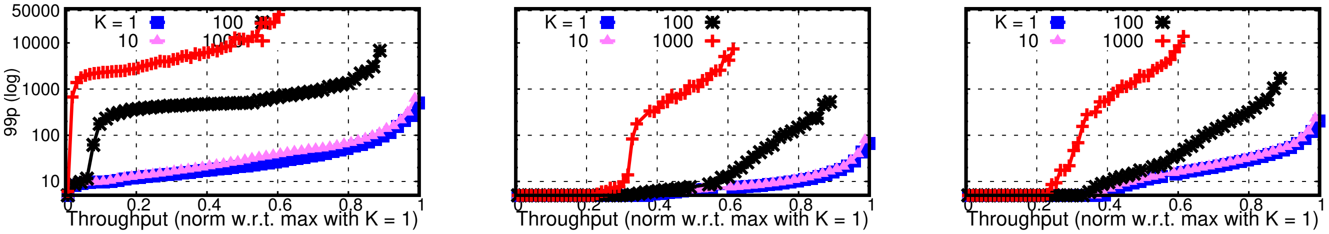
2.2 Variations in Item Size and Tail Latencies

Variations in item size have profound implications for tail latencies. As anecdotal evidence, Nishtala et al. report that in the Facebook `memcached` servers the median response time is 333 microseconds, while the 95th percentile is 1.135 milliseconds [55]. In this section we show that this finding goes beyond the anecdotal, and that all common size-unaware sharding techniques exhibit high tail latencies for workloads in which even only a small fraction of requests targets large items. In particular, we show that, even under moderate loads, the (100-N)th percentile is affected dramatically by a fraction, much smaller than N%, of requests for large items. In the following we report on the 99th percentile, commonly used in Service Level Objective (SLO) definitions, but the results apply also to other high percentiles.

We simulate three common size-unaware sharding techniques on a server with 8 cores, each with a queue to store incoming requests[1]:

- **Early binding**: requests are dispatched to a queue for a particular core, often based on a keyhash, similar to what is used, for instance, in the EREW version of MICA [45].
- **Late binding**: requests are kept in a single queue and dispatched to a core when it becomes idle, similar to what is used, for instance, in RAMCloud [57].

[1] The goal of this simulation is *not* to predict quantitatively the performance differences between these strategies in any real implementation, as their performance is affected by factors such as locality, cost of synchronization, and cost of dispatching, which we do not simulate. Our goal is to demonstrate, for all three methods, the substantial increase in tail latency as a result of the presence of a small fraction of requests for large items.



(a) Early binding.

(b) Late binding.

(c) Early binding + work stealing.

Figure 2: Throughput vs. 99th percentile of response times for different types of size-unaware sharding techniques (y axis in log scale). The workload distribution is bimodal: 0.125% of requests is for large items, whose service time is K time units; the remaining is for small ones, whose service time is 1 time unit. K is varied from 1 to 1,000. K = 1 corresponds to a baseline workload with only small requests. A small (<1%) fraction of large requests suffices to hamper greatly the 99th percentile of response times, and to considerably reduce the achievable throughput.

• **Early binding with work stealing**: requests are handled as in the early binding case, but in addition idle cores steal requests from the queues of other cores, similar to what is used, for instance, in ZygOS [58].

For simplicity, we use a workload with a bimodal size distribution. Small requests form 99.875% of the workload, and have a service time of 1 time unit. Large requests form the remaining 0.125%. We run different simulations in which the service time of large requests is, respectively, K = 1, 10, 100 and 1,000 time units. These values are in line with the order-of-magnitude differences in service time between small and large items observed on our platform (see Figure 1). We use K = 1 to establish a baseline where all requests are small. Inter-arrival times follow an exponential distribution.

Figure 2 shows the 99th percentiles for the three sharding strategies under the bimodal workload compared to a workload with an identical offered load, but with only requests for small items. Even though the fraction of large items requested is much smaller than 1%, all three strategies suffer from a considerable increase in the 99th percentile latency. For K = 100 and K = 1,000, at only 10% utilization the 99th percentile for the early binding design is two orders of magnitude higher than the 99th percentile in the workload composed only of small requests. Stealing and late binding are more resilient to service time variability at low load, but at higher loads they also suffer from one or two orders of magnitude degradation of the 99th percentile, with respect to the workload with only small requests.

The reasons for these increases in the 99th percentile latency are different from one strategy to the next. Early binding suffers from head-of-line blocking when a request for a small item ends up in a queue behind a request for a large item, or behind a request for a large item being executed by this core. The late binding of requests to cores is more resilient to head-of-line blocking, a well known result from queueing theory [28], but it does not avoid it. Late binding is vulnerable to cases in which the arrival of many large requests in a short period of time leads many (or even all) cores to be busy serving large requests. Such an event temporarily reduces the amount of resources available to serve small requests, which impacts tail latency. Stealing improves the tail latency of the early binding design, as it steals some of the requests that would otherwise experience head-of-line blocking but it cannot completely avoid head-of-line-blocking. First, stealing only occurs when a core is idle, and the likelihood of a core being idle decreases as the load increases. Second, by the time a core becomes idle, a request that it steals is likely to have already experienced some head-of-line blocking in the queue from which it is stolen.

In light of these results, Minos processes requests for small and large items on disjoint set of cores, a technique we call *size-aware sharding*. This addresses the shortcomings of existing approaches, by avoiding that a small request waits for the completion of a large one.

3 Minos design

3.1 Size-aware sharding

Preliminaries. We consider a server with n cores. The server has a NIC with multiple receive (RX) and transmit (TX) queues. We configure the NIC to use n RX queues and n TX queues. At any time, there are n_l cores handling requests for large items and n_s cores handling requests for small items ($n_l + n_s = n$). With a slight abuse of language, we say that a request for a small (large) item is a small (large) request, and that a core handling small (large) requests is a small (large) core. In addition to an RX and a TX queue, each large core maintains a software queue.

In the following, we assume all n cores are within the same NUMA domain, so that KV item accesses and inter-core communication happen within the same NUMA domain. Minos can seamlessly scale to multiple NUMA domains by running an independent set of small and large cores within each NUMA domain, and by having clients send requests to the NUMA domain that stores the target key [45].

We consider a KV store with the usual CRUD (Create, Read, Update, Delete) semantics. A client can perform a GET(key) and a PUT(key, value). Create and delete are considered special versions of PUT, and not discussed any further. When a client issues GET and PUT operations, the client software puts in the request the id of the RX queue in which the corresponding packets are deposited when they arrive at the server. The target RX queue is chosen at random for GET operations, and depends on the keyhash for PUT operations (as we describe in Section 4.2). A PUT request also includes the size of the item that is being written. The client does not know the size of an item to be read. Furthermore, the client does not need to know which or how many cores on the server handle small or large requests.

In the following discussion we initially assume that we know the threshold on the item size that separates small and large items. We explain later how the threshold is determined. We first explain size-aware sharding with a given number of small cores and one large core. Then, we show how the number of small and large cores is determined, and how the system operates with more than one large core.

Receiving incoming requests. Only the small cores read incoming requests from the RX queues. They do so in batches, to amortize the cost of communicating with the NIC. Each small core repeats the following sequence of actions w.r.t. the RX queues. First, it reads a batch of B requests from its own RX queue. Then it reads a batch of B/n_s requests from the RX queue of the large core. In this way, all RX queues are drained at approximately the same rate. The reason a large core never reads incoming requests from its RX queue is that, if it were to receive a small request, this request could experience head-of-line blocking behind large requests.

We start by explaining how GET operations are handled.

Operation of the small cores. For each request, a small core starts processing the request by looking up the item associated with the requested key. If its size is below the threshold, the small core continues the GET operation and replies to the client with the requested item (by putting the corresponding reply packet(s) on its TX queue). Else, the small core puts the request in the software queue of the large core.

Operation of a large core. For each request in its software queue, a large core finds the corresponding item, and replies to the client by putting the reply packet(s) on its TX queue.

The operation of a PUT is mostly similar, except that the size is present in the request. Hence, there is no need to do a lookup to find the size. Depending on the size, the request is handled either immediately by the small core or passed on by the small core to the large core, and handled there.

How to find the threshold between large and small requests. Each small core maintains a histogram of the number of requests that correspond to item sizes in certain ranges. Each range corresponds to a size *class*. This histogram is updated on the receipt of every request according to the size of the target item. Periodically, core 0 aggregates these histograms, finds the size class corresponding to the Nth percentile of item sizes, declares that class to be the threshold for the next epoch, and resets the histograms.

To be resilient to workload oscillations, core 0 smooths the values in the aggregated histogram (noted H) according to a moving average that uses the histogram obtained in the previous epoch (noted H_{curr}). For each entry i, core 0 computes $H_{curr}[i] = (1-\alpha)H_{curr}[i] + \alpha H[i]$, and uses the new H_{curr} to determine the Nth percentile. α is a discount factor in the range [0,1], and determines the weight of the new measurements over previous ones. Because Minos targets high throughput workloads, many requests are sampled during an epoch. Hence, H is highly representative of the current workload, and is assigned a weight equal to 0.9 [65].

How to choose the number of small cores. Minos maintains a cost function that gives us for a request of a given size a certain processing cost. Minos can use various cost functions, but currently uses the number of network packets handled to serve the request as cost, either the number of packets in an incoming PUT request or the number of packets in an outgoing GET reply. Alternatives could be the number of bytes or a constant plus the number of bytes. The number of small cores is then set to the ceiling of the fraction of the total processing cost for small requests times the total number of cores. The remaining cores are used as large cores.

Operating with a number of large cores different from one. If, as a result of the above calculation, there is more than one large core, then Minos distributes the large requests over the large cores such that each large core handles a non-overlapping contiguous size range of requests, and such that the cumulative processing cost of requests assigned to each large core is the same. By doing so, not only does Minos balance the load on large cores, but it also shards large requests in a size-aware fashion. That is, the smallest among the large requests are assigned to the first large core, and larger requests are progressively assigned to other cores. A small core that receives a large request puts the request in the software queue of the large core that is handling the size of the requested item.

If all cores are deemed to be small cores, then one core is designated a standby large core. In other words, it handles small requests, but if a large request arrives, it is sent to this core, which then becomes a large core.

3.2 Discussion

Design rationale. The goal of Minos is to improve the Nth percentile. To that end, Minos identifies the smallest N percent of the requests, and isolates the processing of these requests from the processing of larger requests, such that no head-of-line blocking occurs. Furthermore, Minos assigns a number of cores to small/large requests proportionally to the expected load generated by requests of that size, so as to balance the load across cores.

The use of randomization and of the hashed value of the key to decide the target RX queue for a request leads to reasonable load balance among the RX queues. A similar observation was made in the context of MICA [45]. Since the small cores handle the requests that arrive in their own RX queue, and an equal portion of the requests that arrive in the RX queues of the large cores, overall the load is balanced among the small cores. By using purely hardware dispatch for the small requests we eliminate any unnecessary overhead in their processing, such as, for instance, software dispatches. We achieve these results while never dropping large requests, since there is always at least one core available for handling large requests.

The only overheads compared to a purely hardware dispatch solution such as MICA are then: 1) software dispatch for the very small number of large requests, 2) synchronization on the RX queue and the software queue of the large cores, for which we found contention to be low, and 3) some minor loss in locality for the small requests that arrive in the RX queues of large cores.

Not sharding small requests. Minos could implement size-aware sharding for small requests. This would allow for isolating requests of different sizes at a finer granularity. Minos eschews this design choice because it targets SLOs expressed in terms of a single response time percentile. Hence, it is less important to further improve the performance of smaller requests than to achieve the highest throughput with low target tail latency. Sharding small requests across multiple class sizes, instead, may result into a less efficient design because small cores would spend much of their resources in dispatching requests that are served by other cores, and may be idle while waiting to receive dispatched requests. We have experimented with a design in which we shard small requests, and it proved to perform poorly. Assigning all small requests to the same set of cores allows Minos to perform software handoff only for the few large operations, and to achieve high throughput and low Nth percentile latency.

Target percentile setting. The latency benefits brought by Minos naturally depend on the setting of the target percentile and the item size distribution. For example, an item size distribution could be such that the 95th percentile is 10B, the 96th percentile is 500KB and the 99th percentile is 1MB. Then, optimizing for the 95th percentile would benefit the latency of the smallest 95% of operations more than what would happen for the smallest 99% of operations if Minos was set to optimize for the 99th percentile. However, if the target SLO of the application using the key value store is expressed in terms of the 99th percentile, it is less important to achieve a very good 95th percentile by separating 10B requests from the rest, and Minos should be configured to target the 99th percentile. In this setting, Minos would improve the 99th percentile latency as much as possible by segregating 1MB operations and larger ones from the rest.

In the current design, Minos takes the target percentile as input. The system administrator may determine such percentile with the aid of workload traces collected offline, which are typically available in production systems [3, 5, 59]. Automatically determining a suitable percentile that results in high latency gains and high throughput is an orthogonal research issue that we are currently investigating.

Trade-offs. In Minos small and large operations each have access only to a subset of the processing power available on the machine. This may lead some requests to experience a longer queueing time than what they would experience if the request could be served by any core. The impact of this additional delay on short requests is outweighed by the benefits that stem from avoiding head-of-line blocking. This design, however, penalizes large requests. The rationale underlying this trade-off is that Minos aims to reduce a target Nth percentile of the response time distribution by favoring the smallest N% of the operations. Larger requests that fall out of such percentile, then, are processed in a best effort fashion –and, importantly, never dropped.

Penalizing larger request is an inevitable price to pay to favor smaller ones, as shown by the theoretical and quantitative analysis of scheduling policies similar to size-aware sharding [1, 6, 18]. We assess the effects of this trade-off on performance in Section 6.1, and we discuss the differences between size-aware sharding in Minos and related scheduling techniques in Section 7.

Alternative designs. We now discuss alternative designs to address item size variability, and why we do not adopt them.

1) *Use a dedicated set of machines to serve large requests*, as suggested in [45]. This solution may lead to waste of resources because the workloads of large and small requests cannot be consolidated. It also requires migrating items across machines in case an item changes size, and adds one network hop to redirect large requests.

2) *Splitting large operations in smaller chunks*. This allows interleaving the processing of such chunks with small requests. This design may lead to lower resource efficiency with respect to the run-to-completion model adopted by Minos. First, it may lead to worse data locality, by accessing memory regions corresponding to different requests, and by interleaving request processing with networking operations. Second, it requires the implementation of nontrivial scheduling mechanisms, whose costs may be not negligible with μsecond scale SLOs. Instead, the run-to-completion model enables high efficiency [57], and allows us to re-use state-of-the-art techniques proposed for such model [45]. In addition, it allows Minos to avoid head-of-line blocking by implementing a simple FIFO scheduling policy within each core.

4 Implementation

4.1 Network stack

Minos relies on the availability of a multi-queue NIC with support for redirecting, in hardware, a packet to a specific queue on the NIC (e.g., RSS [32] or Flow Director [33]). This feature is now commonplace in commodity NICs.

To reduce packet processing overhead, Minos uses the Intel DPDK library [23] to implement a user-level zero-copy network stack. All memory for the DPDK library is statically allocated and accessible by all cores. Packets are received directly in memory, thus enabling zero-copy packet processing. Furthermore, Minos uses DPDK-provided lockless software rings to dispatch large requests from small to large cores without any copies [39]. Small cores check for incoming requests by means of polling, to avoid costly interrupts [57]. Similarly, large cores use polling to check for incoming requests on their software queue. Requests are moved in batches to further limit overhead.

Clients and servers communicate using UDP, implemented on top of Ethernet and IP. Clients use the UDP header to specify the target RX queue for a given packet. Requests that span multiple frames (large PUT requests and large GET replies) are fragmented and defragmented at the UDP level.

Retransmission is handled by the client. Similar to previous work [45], Minos does not support exactly-once semantics and assumes idempotent operations. Exactly-once semantics can be achieved by means of request identifiers.

4.2 KV store and memory management

Data structures. Minos employs the KV data structures used in MICA [45]. Keys are split in *partitions*. Each partition is a hash table, each entry of which points to a bucket, equal in size to a cache line. Each bucket contains a number of slots, each of which contains a tag and a pointer to a key-value item. A first portion of the keyhash is used to determine the partition, a second portion to map a key to a bucket within a partition, and a third portion forms the tag [22, 45], which is used to reduce the number of random memory accesses when performing a key lookup. Overflow buckets are dynamically assigned to a bucket when it has reached its maximum capacity.

Memory management. The current prototype of Minos employs the memory manager of the DPDK library to handle allocation of memory regions for key-value entries. Minos can be extended to integrate more efficient memory allocators, such as the one based on segregated fits of MICA, or a dynamic one as in Facebook's `memcached` deployment [55].

Concurrency control. Minos uses a concurrency control scheme that is similar to Concurrent Read Exclusive Write (CREW) [45]. Each core is the *master* of one partition, and each key can be written only by the master core of corresponding partition. This serializes write operations on a key.

The concurrency control scheme in Minos differs slightly from CREW, as a result of the distinction between small and large cores. PUTs on keys whose master core is a small core proceed along the lines of CREW. PUTs on keys whose master core is a large core may be served by any core (either because the request is small, or because it is dispatched to a large core different from the one which receives the request). In addition, two concurrent PUT operations on the same key may be assigned to two different cores (a small and a large one, or two large ones), depending on the size of the corresponding values. Hence, all PUTs are guarded by a spinlock.

We argue (and we experimentally show) that the corresponding overhead of spinlocks is largely outweighed by the benefits of size-aware sharding, especially for the read-dominated workloads that are prevalent in production environments [3, 10, 55, 56]. First, in such workloads PUTs are rare. Second, PUTs on large cores proceed mostly without contention, because large cores serve non-overlapping size ranges, so requests for the same large item are sent to the same core. Third, PUTs on small cores mostly proceed without contention because of the CREW nature of the concurrency protocol for keys whose master is a small core.

GETs can be served by any core, and are processed by means of an optimistic concurrency control scheme [45]. Each bucket has a 64-bit epoch, which is incremented when starting and ending a write on a key stored in that bucket. Upon reading, a core looks at the epoch. If it is odd, then there is an ongoing write on a key of the bucket, and the read is stalled until the epoch becomes even. If (or when) the epoch is even, the core saves the current epoch value and performs the read. After the read, the core re-reads the epoch of the bucket. If the value is the same as when the read started, the read is successful. Else, a conflicting write might have taken place, and the read is restarted. Because all memory is pre-allocated, a writer thread can safely modify/erase a KV entry that is concurrently accessed by a read.

5 Experimental Platform

5.1 Hardware

Our platform is composed of 8 identical machines equipped with an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz with 8 physical cores and 64 GB of main memory. The machines run Ubuntu 16.04.2 with a 4.4.0-72-generic kernel. One machine acts as server and the other 7 run the client processes. We disable hyperthreading and power-saving modes on all the machines. All the machines are equipped with a 40Gbit Mellanox MT27520 NIC (ConnectX-3 Pro), are located in the same physical rack, and are connected via a top-of-rack switch. The network stack relies on the Intel DPDK library (version 17.02.1), to which we allocate 50 1GB huge pages.

Our NIC supports only RSS to implement hardware packet-to-RX queue redirection [50]. RSS determines the RX queue for an incoming packet by performing the hash of the quintuplet composed of source and destination IP, source and destination port and the transport layer protocol. To allow the clients and the server to send packets to specific RX queues, we ran a set of preliminary experiments to determine to which port to send a packet so that it is received by a specific RX queue. More flexible hardware packet redirection methods can be used on NICs that support them. For example Minos can use Flow Director [33, 49] to set the target RX queue as UDP destination port of a packet.

5.2 Systems used in comparison

We compare Minos with three systems that implement state-of-the-art designs of KV store, and that are based on the queueing models that we have described in Section 2.

• **Hardware Keyhash-based sharding (HKH).** This system implements early binding of requests to cores, as done in MICA [45]. Requests are redirected in hardware to the target core, according to the CREW policy. This policy performs the best on skewed read-dominated workloads [45], such as our default workload.

• **Software hand-off (SHO).** This system implements the late binding of requests to cores, as in RAMCloud [57]. SHO uses disjoint sets of handoff and worker cores. Each handoff core has a software queue, in which it deposits the requests taken from its RX queue. Worker cores pull one request at a time from the handoff queues, process the corresponding KV request, and reply to the client. The best number of handoff cores depends on whether the workload is CPU or network bound. We have experimented with 1,2 and 3 handoff cores. We report experimental results corresponding to the best configuration for each workload.

• **HKH + work stealing (HKH+WS).** This system implements request stealing on top of HKH, as in ZygOS [58]. Each core has a software queue where it places the requests taken from its own RX queue. An idle core can steal requests from the software queues of other cores, and from their RX queues, if no request is found in any software queue.

All designs are implemented in the same codebase. This allows us to focus on the effects of item size heterogeneity on performance, and to factor out implementation differences (e.g., in the KV store data structure and concurrency control scheme) and limitations (e.g., leak of support for multi-frame packets and additional overheads to support richer APIs) of the existing systems that implement the designs we consider.

The internal parameters of Minos are set as follows. Workload statistics are collected by core 0 every second. The byte range corresponding to the i-th size class is $[2^{(i-1)}, 2^i)$, and i ranges from 1 to 10. The size of a batch of requests read from a RX queue is 32, and the same batch size is used for other systems as well.

% large reqs (p_L)	Max size (s_L)	% data for large reqs
0.125	250 KB	25
	500 KB	40
	1000 KB	60
0.0625	500 KB	25
0.25		60
0.5		75
0.75		80

Table 1: Item size variability profiles.

5.3 Workloads

We use workloads characterized by different degrees of item size variability and GET:PUT ratios.

Item size variability. We use, as a starting point, the characterization of the ETC workload at Facebook [3]. Specifically, we consider a trimodal item size distribution, according to which an item can be tiny (1-13 bytes), small (14-1400 bytes) or large (1500-maximum size). The size of an item within a class is drawn uniformly at random. To generate workloads with different degrees of item size variability, we vary both the percentage of large requests, (noted p_L), and the size of items corresponding to large requests, by changing the maximum size of large items (noted s_L). We let s_L range from 250KB to 1MB. These values are consistent with the production workloads we discussed in Section 2.1. Similarly to what is seen for the ETC workload, we set $p_L < 1\%$, so that the 99th percentile of the requests service times corresponds to small and tiny items only. Specifically, we vary p_L from 0.0625 to 0.75. Table 1 reports the combinations of p_L and s_L we consider. It also reports the corresponding percentage of bytes that are exchanged because of large requests.

Key popularity. We consider a skewed workload that follows a zipfian distribution with parameter 0.99. This represents the default value in YCSB [14], is widely used in the evaluation of several KV stores [45, 35], and is representative of the strong skew of many production workloads [3].

We use the zipfian distribution on the sets of tiny and small items, because they are many and they exhibit small variability in size. Large items, instead, are much fewer and exhibit much higher variability, and are therefore chosen uniformly at random. This avoids pathological cases in which the most accessed large item is the biggest or the smallest item, thereby skewing the results.

We consider a dataset of 16M key-value pairs, out of which 10K are large elements. Of the remaining key-value pairs, 40% correspond to tiny items, and 60% to small ones. This setting is consistent with the item size distribution and the low access probability of individual large keys that characterize the ETC workload. Each large item has, in fact, a probability $p_L/100 \cdot 10K/16M$ of being accessed. For simplicity, we keep the size of the keys constant to 8 bytes.

Write intensity. We consider a read-dominated and a write-intensive workloads, corresponding, respectively, to a 95:5 and 50:50 GET:PUT ratio. These values are used as default values in YCSB and KV store evaluations [45, 35] (the ETC workload has a 97:3 GET:PUT ratio).

Default workload. We set a default value for each parameter, and generate different workloads by changing the value of one parameter at a time while keeping the other ones to their default values. The default workload is skewed with a 95:5 GET:PUT ratio, a percentage of large requests equal to 0.125 and a maximum large item size of 500 KB.

5.4 Benchmarking methodology

Load generation. We spawn 8 threads per client machine, each pinned to a separate physical core and to an RX queue. Client threads simulate an open system by generating requests at a given rate, which varies depending on the target throughput. The time between two consecutive requests of a thread is exponentially distributed.

Measurements. Each request is timestamped with the send time at the client, which is piggybacked by the server on the reply message. Client threads constantly check their own RX queues for replies, and compute the end-to-end latency of a request using the timestamp in the reply message.

A client thread can have multiple requests in flight, so for simplicity packet retransmission is not enabled. For this reason, we only report performance values corresponding to scenarios in which the packet loss rate is equal to 0.

Each workload runs for 60 seconds. The first and last 10 seconds are not included in the reported results.

Performance metrics. We focus on maximum achievable throughput (number of of successful operations completed per second) and 99th percentile of end-to-end latencies, since large requests correspond to less than 1% of the total. We also measure the utilization of the server NIC to evaluate whether Minos is able to fully use the available bandwidth.

We consider SLOs in the form "The 99th percentile of latencies must be within X μsec". We use X = 50 and X = 100 to evaluate the performance gains of Minos as a function of the strictness of the SLO. These values correspond to 10 and 20 times the mean service time for a GET request in our default workload (similarly to previous work [58]).

6 Evaluation

6.1 Default workload

Throughput vs. 99th percentile latency. Figure 3 shows the 99th percentile latency (99p) as a function of the throughput with the default workload. Minos achieves the highest peak throughput (6.2 Mops) and the lowest latency ($\leq 50\mu sec$ up to 90% of peak throughput).

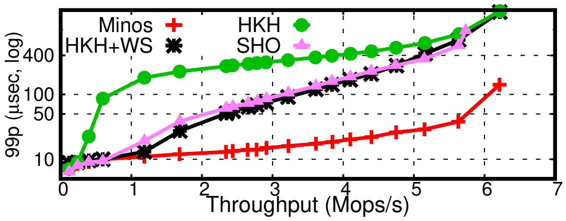


Figure 3: Throughput vs. 99th percentile latency (y axis in log scale) with the default workload. Minos matches the throughput of the purely hardware-based design and achieves the lowest latency.

Minos achieves the same peak throughput as HKH and HKH+WS, reflecting the fact that all three systems rely mostly or entirely on hardware handoff for request distribution (at very high load, stealing in HKS+WS rarely happens). SHO achieves 10% less peak throughput, because it is bottlenecked by the software handoff. In terms of 99th percentile, Minos does better than HKH at any load, with improvements reaching one order of magnitude as soon as the load exceeds 1 Mops. HKH+WS and SHO start out with similar 99th percentile latencies as Minos under loads below 1 Mops, but under high load their 99th percentile latencies rapidly deteriorate to reach values similar to HKH. For an SLO on the 99th percentile latency of 50 μsec Minos can perform 5.6 Mops, 2.4 times the throughput of its best competitor (HKH+WS). For an SLO of 100 μsec, Minos still achieves 1.75 times the throughput of its best competitor.

Minos achieves the best performance by overcoming the limitations of existing designs when dealing with variable-size items, and that we have discussed in Section 2.2. Interestingly, the performance curves of the competitor systems we consider follow the ones depicted in Figure 2, which portrays the behavior of the same systems in idealized conditions (i.e., without dispatching and synchronization costs). This indicates that the reason for the worse 99th percentile tail latency exhibited by such systems is primarily due to the shortcomings of their designs in presence of item size variability, and not to low level implementation details.

Latency of large requests. Minos leverages the insight that the latency of the largest N% of the requests should not impact the (100-N)th percentile. Minos restricts the N% largest requests to a subset of the cores (N=1 in our setting), which may result in increased latencies for such requests. We now evaluate the performance penalty incurred by large requests in Minos as a consequence of size-aware sharding. Figure 4 reports the 99th percentile latency of large requests in Minos and HKH+WS (the best alternative).

Inevitably, Minos imposes some penalty on the performance of large requests, reaching up to a factor of 2 for the 99th percentile latency of large requests before the system goes into saturation. We argue that moderately penalizing

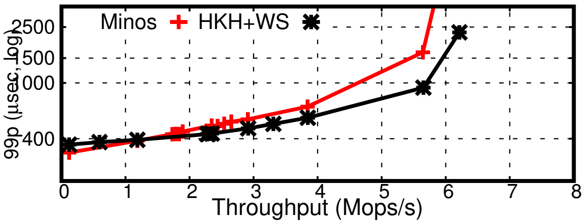


Figure 4: Throughput vs. 99th percentile latency of large requests with the default workload (y axis in log scale). Minos trades its large benefits in terms of the overall 99th percentile for a moderate penalty on the large requests, which represent a small fraction of the workload.

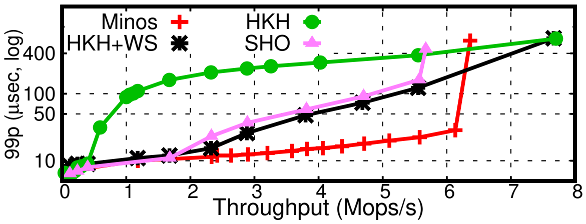


Figure 5: Throughput vs. 99th percentile latency for Minos vs. existing designs with the 50:50 GET:PUT workload (y axis in log scale).

large requests is a reasonable price to pay for the order-of-magnitude improvement for the target (100-N)th percentile.

Minos can improve the latency of large requests by allocating more cores to them. Minos currently determines the number of small cores by taking the ceiling of the total number of cores times the fraction of load generated by small requests. For this workload, it allocates only one core to the large requests. This represents an over-allocation to small requests to completely isolate them from large requests, and hence an under-allocation for large requests. An alternative strategy is to allocate one more core to large requests, and let large cores steal from the RX queues of small ones to fully use any extra capacity. To avoid re-introducing head-of-line blocking, stealing can be done one request at a time, so that there is never a small request queued behind a large request. We are currently experimenting with this alternative design, which would improve performance for large requests, while only introducing a small degradation for small requests.

6.2 Write-intensive workload

We now investigate the effect of write intensity on Minos. Figure 5 reports the 99th percentile of response times with all four systems and a 50:50 GET:PUT workload.

Minos continues to deliver a 99th percentile latency one order of magnitude lower than alternative approaches, up to the saturation point at 6.3 Mops, but overall achieves a lower (by 10%) throughput than HKH and HKH+WS. Throughput values are in general higher than with the 95:5 workload, because replying to a PUT requires less network bandwidth, since the response message does not contain any item value payload. This behavior is consistent with that observed by previous work [45]. SHO is the only exception, as handoff cores represent the bottleneck.

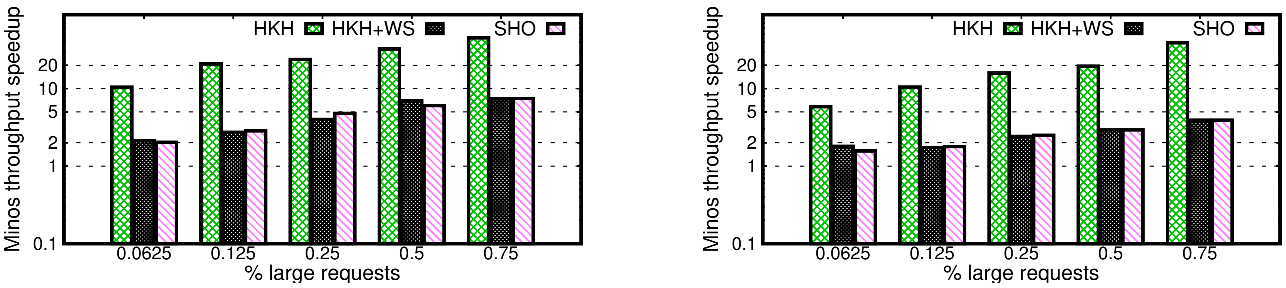
Minos achieves a lower throughput with respect to HKH and HKH+WS because of the overhead stemming from profiling the workload and periodically aggregating them on core 0 to compute the 99th percentile of the item sizes. We are currently investigating techniques to reduce such overhead, e.g., sampling only a subset of the requests. Alternatively, the threshold between large and small requests can be set statically if it does not vary over time and traces of the target workload are available for off-line analysis (as typical in production workloads [3, 55, 59]). With this variant, Minos is able to match the throughput of HKH and HKH+WS.

6.3 Sensitivity to item size distribution

We vary the percentage of large requests in the workload (p_L) and the maximum size of large requests (s_L). When changing the value of one, the other parameter keeps the default value. We then measure the maximum throughput achievable under the two SLOs we consider.

Figure 6 and Figure 7 report the increase in throughput achieved by Minos compared to the other designs (y axis in log scale). Figure 6 shows the results of the experiments in which we change p_L. Figure 7 refers to changing s_L. The graph on the left uses an SLO of 50 μsec, the one on the right 100 μsec. When varying p_L, the maximum throughput achieved by Minos within the $50\mu sec$ ($100\mu sec$) SLO ranges from 6.2 to 1.7 Mops (6.9 to 2.3 Mops), corresponding to $p_L = 0.0625$ and $p_L = 0.75$. When varying s_L, the maximum throughput achieved by Minos within the $50\mu sec$ ($100\mu sec$) ranges from 6.2 to 4.7 Mops (6.9 to 4.7 Mops), corresponding to $s_L = 250KB$ and $s_L = 1000KB$.

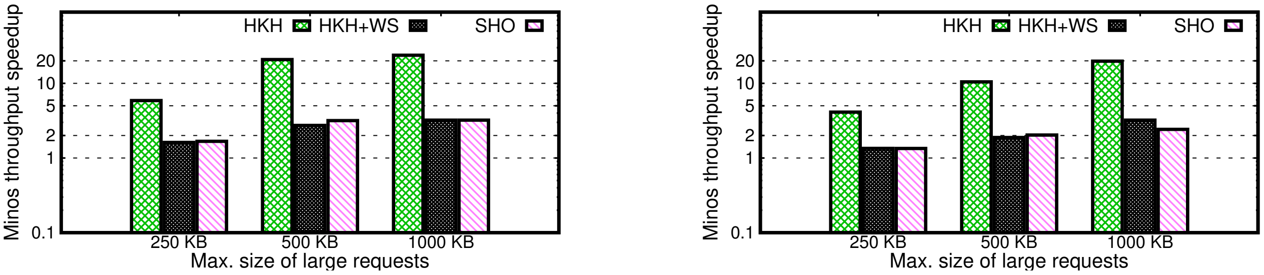
Minos outperforms existing designs, achieving consistently higher throughput for a given workload and a given SLO. The throughput speedup grows with p_L and s_L, because the increased presence of large(r) requests negatively affects the latency of small requests, and hence the 99th percentile, in alternative designs. As expected, the throughput gains are higher with the stricter SLO: the looser is the performance target, the smaller is the impact of Minos' design. For the stricter SLO, Minos achieves a speedup of up to 7.4 w.r.t HHK+WS (corresponding to the $p_L = 0.75$ case), i.e., the second best design. For the looser SLO, the speedup ranges from 1.34 ($s_L = 250$KB) to 3.9 ($p_L = 0.75$).



(a) 99p ≤ 50μsec.

(b) 99p ≤ 100μsec.

Figure 6: Maximum throughput achievable for a given 99th percentile latency SLO with different percentages of large requests (y axis in log scale). Each bar represents the speedup of Minos over an alternative design (higher is better).



(a) 99p ≤ 50μsec.

(b) 99p ≤ 100μsec.

Figure 7: Maximum throughput achievable for a given 99th percentile latency SLO with different maximum sizes of large requests (y axis in log scale). Each bar represents the speedup of Minos over an alternative design (higher is better).

6.4 Higher network bandwidth

With the default workload, the NIC is 93% utilized. With higher percentages of large requests, the system becomes even more network-bound. In this section we investigate whether Minos can take advantage of larger network bandwidths. Because we cannot provision our machines with more bandwidth, we relieve the NIC bottleneck by sampling the number of replies that the server sends back to clients. That is, the server processes requests as before, up to the time at which it would otherwise send the reply to the client. Then, instead, it only sends replies to a percentage *S*% of the total requests, and drops the remaining ones. We vary *S* from 100 to 25, and we measure the achieved performance (throughput and 99th percentile latency), as well as the utilization of the NIC. We choose the read-intensive workload with $p_L = 0.75$, as it quickly saturates the NIC with $S = 100$.

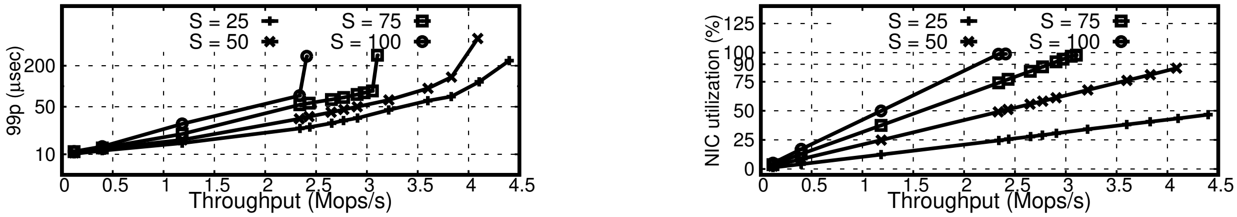
Figure 8 reports the results of the experiment. The left plot shows the throughput vs. 99th percentile latency (y axis in log scale). The right one shows the utilization of the NIC as a function of the throughput. As *S* decreases, Minos can sustain higher loads, because the bottleneck is increasingly shifted towards the CPU. Minos is able to fully utilize the available resources, by reaching throughput values that saturate (or almost saturate) the NIC (S = 100,75,50) except when the bottleneck is query processing (S = 25).

6.5 Load balancing

We now evaluate the ability of Minos to distribute the load evenly across cores according to the provided cost function. To this end, we measure the load sustained by each core with p_l = 0.0625, 0.25, 0.75, corresponding to low, medium and high load posed by large requests. Figure 9a reports the percentage of requests performed, and Figure 9b reports the percentage of packets processed by each core (y axis in log scale). Two conclusions can be drawn. First, all cores process roughly the same number of packets, and hence roughly perform the same amount of work. Small cores obviously process more requests per second, as these requests involve less work. Large cores process different requests per second among each other, as a consequence of the size-aware sharding that Minos implements also within large requests. Second, Minos varies the number of small and large cores as a function of the workload, such that enough resources are allocated to small and large requests.

6.6 Dynamic workload

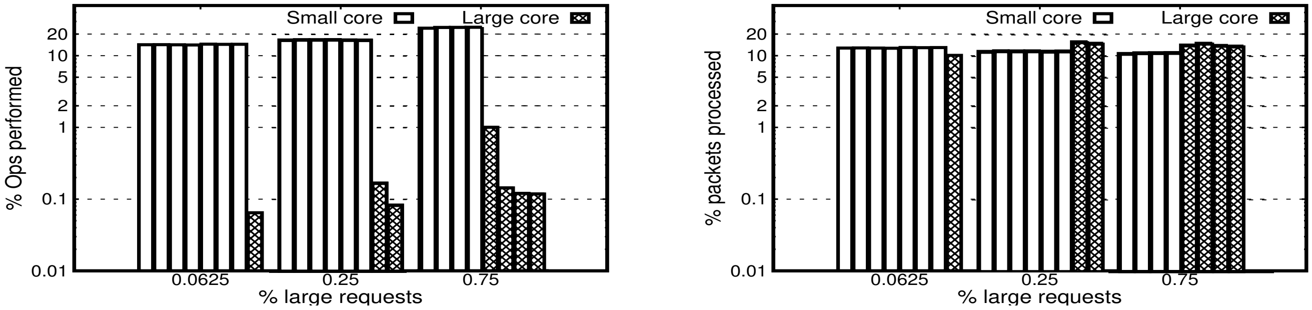
We finally demonstrate the capability of Minos to adapt to changing workloads. To this end, we run a workload in which the percentage of large operations p_L varies every 20 seconds. It first grows gradually from 0.125 to 0.75, and



(a) Throughput vs. 99th percentile latency

(b) Throughput vs. NIC utilization.

Figure 8: Scalability of Minos with more network bandwidth ($p_L = 0.75$). S is the sampling percentage used to simulate more network bandwidth. Minos processes and replies to S% of the requests. The remainder is processed, but the reply is dropped. Minos scales with more bandwidth (a) and saturates the NIC (b), except when query processing is the bottleneck ((b), S = 25).



(a) **Operations** per second.

(b) **Packets** per second.

Figure 9: Breakdown of the load per core in Minos (y axis in log scale). Large cores process fewer requests per second than small cores (a), but the number of packets processed per second is uniformly distributed across cores (b).

then shrinks back to 0.125. We keep the request arrival rate fixed at 2.25 Mops, corresponding to high load for $p_L = 0.75$. Figure 10(top) compares the performance achieved by Minos and HKH+WS, i.e., the second best design. Each point represents the 99th percentile latency as measured over a 1 second window (y axis in log scale). Figure 10(bottom) shows how many cores Minos assigns to large requests over time. Minos achieves latencies up to 20 times lower than HKH+WS (70 μsec vs $\approx$ 1.5 msec with $p_L = 0.75$). Minos achieves this result by programmatically allocating cores to small and large requests proportionally to their corresponding loads.

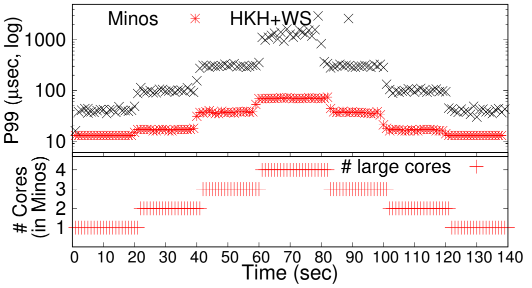


Figure 10: Evolution over time of the 99th percentile latency of Minos and HKH+WS with a dynamic workload (top, with y axis in log scale) and evolution over time of number of large cores in Minos (bottom). Every 20 seconds the percentage of large requests changes, first growing from 0.125 to 0.75 and then shrinking back. Minos adapts to changing workload conditions and delivers up to 20X lower 99th percentile latencies.

7 Related Work

To the best of our knowledge, Minos is the first KV store to introduce the concept of size-aware sharding to address the challenges of delivering μsec-scale tail latency in presence of item size variability. We now discuss related systems.

In-memory KV stores. A plethora of in-memory KV stores have been proposed in the last years. These systems propose different designs based on new data-structures (CPHash [52], Masstree [48], MemC3 [22]) and lightweight network stacks (Chronos [37], MICA [43, 45], RamCloud [57], RockSteady [40]), or on the use of RDMA (Pilaf [53], Herd [35], FaRM [21], RFF [60], FaSST [36], TailWind [61]), FPGAs (KV-Direct [41]), GPUs (Mega-KV [64], MemcacheGPU [30]), HTMs (DrTM [11, 63]), or other specialized hardware ([38, 9]).

None of these systems addresses the problem of achieving low tail latency in presence of item size variability, which is the primary focus of Minos. In addition, Minos only assumes the availability of commodity hardware. Investigating the synergies between the design of Minos and specialized hardware is an interesting avenue for future work.

Size-aware data-stores. We are aware of a few data stores that take into account the size of items or requests to improve performance. Rein [59] supports multi-key get requests and processes them taking into account the number of keys involved in a request. Rein relies on the assumption that there is only a weak correlation between the size of an item and the service time of a request for that item. Minos, instead, targets workloads with high item size variability, for which the service time of a request strongly depends on the size of the corresponding item (see Figure 1).

AdaptSize [8] is a caching system that reduces the probability of caching large objects, so as to increase the hit rate of smaller, more frequently accessed ones. AdaptSize targets a problem that is orthogonal to Minos, which assumes the presence in memory of both small and large items.

The systems in [12, 29, 65] target static content and leverage a central component (the Linux kernel on a single-core architecture [29] or a scheduler in a distributed system [12, 65]) to implement request scheduling. By contrast, Minos deals with mixed read/write workloads and targets multi-core architectures with multi-queue NICs.

Operating systems. IX [7] and ZygOS [58] use lightweight network stacks to meet μsec-scale SLOs. ZygOS uses work stealing to avoid core idleness and reduce head-of-line blocking. As we show by means of simulation (§ 2.2) and experimental data (§ 6), this approach cannot fully avoid head-of-line blocking as done by Minos, because work stealing *i*) is agnostic of the CPU time corresponding to serving a request; and *ii*) is only triggered by idle cores, whose presence becomes less likely as the load increases.

Scheduling systems. There is a vast literature on scheduling requests with heterogeneous service demands. Several approaches have been applied in the context of flow scheduling [1, 4, 24, 25, 31, 54], single-server request scheduling [26, 42, 47] and cluster request scheduling [18, 20, 19]. Proposed approaches include workload partitioning [16, 18, 29], preempting [6, 19] or migrating requests [26, 27], and stealing [20, 42]. One common result of these approaches is that favoring small requests inevitably comes at the expense of the performance of the largest requests.

Size-aware sharding draws from these techniques, and makes the same trade-off between the latencies of small and large requests. However, Minos substantially deviates from these systems, to apply size-aware sharding in an in-memory key value store efficiently. In particular,

- Minos does not rely on any *a priori* information on the size of a request. This contrasts with existing systems that rely on request runtime estimates, such as Hawk [20].
- Minos avoids head-of-line blocking by processing short and large requests on disjoint sets of cores. This contrasts with systems like 2DFQ [47], where all resources are shared between short and large requests, and hence a burst of large requests may delay shorter ones.
- The design of Minos is tailored for the in-memory key value store domain. *i*) Minos integrates size-aware sharding with the run-to-completion model, which avoids interrupts and context switches, enhances locality, and reduces cache pollution [45, 57]. This is unlike the aforementioned systems, which target the classic multi-threaded approach and migrates requests across cores [26, 42, 44], or across servers [27]. *ii*) Minos leverages the hardware request-to-core dispatching enabled by multi-queue NICs to reduce the amount of software hand-offs. This allows Minos to achieve throughput values equal or close to those achievable by pure hardware request-to-core dispatching. *iii*) Minos co-designs size-aware-sharding and the concurrency control scheme to both achieve load balance and avoid head-of-line blocking.

These characteristics allow Minos to target μscale tail latencies, whereas the aforesaid scheduling approaches reportedly support SLOs in the order of the milliseconds or higher.

8 Conclusion

This paper presents Minos, an in-memory key-value store designed to deliver μsec-scale tail latency with workloads characterized by highly variable item sizes, as frequent in production workloads. Minos implements size-aware sharding, a new technique that assigns small and large requests to disjoint set of cores. This ensures small requests never wait due to the collocation with a long request. Minos identifies at runtime the size threshold between long and short requests, and the amount of cores to allocate to them. We compare Minos to three state-of-the-art designs and we show that, compared to its closest competitor, Minos achieves a 99th percentile latency that is up to 20 times lower. Put differently, for a given value for the 99th percentile latency equal to 10 times the mean service time, Minos achieves a throughput that is up to 7.4 times higher.

Acknowledgements

We thank the anonymous reviewers, Richard L. Sites and our shepherd Amar Phanishayee for their feedback. We also thank Hyeontaek Lim for his help in setting up the DPDK library. This research has been supported by an EcoCloud post-doctoral research fellowship.

References

[1] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th*

USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 19–19.

[2] Arlitt, M. F., and Williamson, C. L. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw. 5*, 5 (Oct. 1997), 631–645.

[3] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proc. of SIGMETRICS* (2012).

[4] Bai, W., Chen, L., Chen, K., Han, D., Tian, C., and Wang, H. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 455–468.

[5] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 363–375.

[6] Bansal, N., and Harchol-Balter, M. Analysis of srpt scheduling: Investigating unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2001), SIGMETRICS '01, ACM, pp. 279–290.

[7] Belay, A., Prekas, G., Primorac, M., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst. 34*, 4 (Dec. 2016), 11:1–11:39.

[8] Berger, D. S., Sitaraman, R. K., and Harchol-Balter, M. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 483–498.

[9] Blott, M., Liu, L., Karras, K., and Vissers, K. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems* (Berkeley, CA, USA, 2015), HotStorage'15, USENIX Association, pp. 8–8.

[10] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 49–60.

[11] Chen, Y., Wei, X., Shi, J., Chen, R., and Chen, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 26:1–26:17.

[12] Ciardo, G., Riska, A., and Smirni, E. Equiload: a load balancing policy for clustered web servers. performance evaluation. *In Performance Evaluation 46* (2001), 46–101.

[13] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 321–334.

[14] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[15] Crovella, M. E., and Bestavros, A. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Trans. Netw. 5*, 6 (Dec. 1997), 835–846.

[16] Crovella, M. E., Harchol-Balter, M., and Murta, C. D. Task assignment in a distributed system (extended abstract): Improving performance by unbalancing load. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1998), SIGMETRICS '98/PERFORMANCE '98, ACM, pp. 268–269.

[17] Dean, J., and Barroso, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[18] Delgado, P., Didona, D., Dinu, F., and Zwaenepoel, W. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 497–509.

[19] Delgado, P., Didona, D., Dinu, F., and Zwaenepoel, W. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the Ninth ACM Symposium on Cloud Computing* (2018), SoCC '18.

[20] Delgado, P., Dinu, F., Kermarrec, A.-M., and Zwaenepoel, W. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 499–510.

[21] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[22] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 371–384.

[23] Foundation, T. L. Data plane development kit. https://dpdk.org, 2017.

[24] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., and Crowcroft, J. Queues don't matter when you can jump them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 1–14.

[25] Guo, L., and Matta, I. The war between mice and elephants. In *Proceedings of the Ninth International Conference on Network Protocols* (Washington, DC, USA, 2001), ICNP '01, IEEE Computer Society, pp. 180–.

[26] Haque, M. E., He, Y., Elnikety, S., Nguyen, T. D., Bianchini, R., and McKinley, K. S. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, ACM, pp. 625–638.

[27] Harchol-Balter, M. Task assignment with unknown duration. *J. ACM 49*, 2 (Mar. 2002), 260–288.

[28] Harchol-Balter, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. Cambridge University Press, New York, NY, USA, 2013.

[29] Harchol-Balter, M., Schroeder, B., Bansal, N., and Agrawal, M. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst. 21*, 2 (May 2003), 207–233.

[30] Hetherington, T. H., O'Connor, M., and Aamodt, T. M. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 43–57.

[31] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 127–138.

[32] Hudek, T. Introduction to receive side scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[33] Intel. Intel 82599 10 gigabit ethernet controller: Datasheet. https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html, 2014.

[34] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 121–136.

[35] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[36] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 185–201.

[37] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 9:1–9:14.

[38] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81.

[39] Kit, D. P. D. Ring library. http://dpdk.org/doc/guides/prog_guide/ring_lib.html, 2017.

[40] Kulkarni, C., Kesavan, A., Zhang, T., Ricci, R., and Stutsman, R. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 390–405.

[41] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.

[42] Li, J., Agrawal, K., Elnikety, S., He, Y., Lee, I.-T. A., Lu, C., and McKinley, K. S. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2016), PPoPP '16, ACM, pp. 14:1–14:13.

[43] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 476–488.

[44] Li, X., Sethi, R., Kaminsky, M., Andersen, D. G., and Freedman, M. J. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016).

[45] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.

[46] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch, T. F. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 36–47.

[47] Mace, J., Bodik, P., Musuvathi, M., Fonseca, R., and Varadarajan, K. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 144–159.

[48] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 183–196.

[49] Mellanox. Mellanox connectx-3 product brief. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf, 2013.

[50] Mellanox. Mellanox dpdk release notes (v 16.11.1.5. http://www.mellanox.com/related-docs/prod_software/, 2017.

[51] Memcached. memcached. http://www.memcached.org.

[52] Metreveli, Z., Zeldovich, N., and Kaashoek, M. F. Cphash: A cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 319–320.

[53] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[54] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 221–235.

[55] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Proc. of NSDI* (2013).

[56] Noghabi, S. A., Subramanian, S., Narayanan, P., Narayanan, S., Holla, G., Zadeh, M., Li, T., Gupta, I., and Campbell, R. H. Ambry: Linkedin's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 253–265.

[57] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[58] Prekas, G., Kogias, M., and Bugnion, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 325–341.

[59] Reda, W., Canini, M., Suresh, L., Kostić, D., and Braithwaite, S. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 95–110.

[60] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 1–15.

[61] TALEB, Y., STUTSMAN, R., ANTONIU, G., AND CORTES, T. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[62] WANG, F., XIN, Q., HONG, B., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MCLARTY, T. T. File system workload analysis for large scientific computing applications. In *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)* (Apr. 2004), p. 139152.

[63] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[64] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow. 8*, 11 (July 2015), 1226–1237.

[65] ZHANG, Q., RISKA, A., SUN, W., SMIRNI, E., AND CIARDO, G. Workload-aware load balancing for clustered web servers. *IEEE Trans. Parallel Distrib. Syst. 16*, 3 (Mar. 2005), 219–233.