The Impact of Thread-Per-Core Architecture on Application Tail Latency

Enberg, Pekka

2019

Enberg, P, Rao, A & Tarkoma, S 2019, The Impact of Thread-Per-Core Architecture on Application Tail Latency. in Architectures for Networking and Communications Systems (ANCS),2019 ACM/IEEE Symposium on. IEEE, pp. 1-8, 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Cambridge, United Kingdom, 24/09/2019. https://doi.org/10.1109/ANCS.2019.8901874

http://hdl.handle.net/10138/313642
10.1109/ANCS.2019.8901874

2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)

The Impact of Thread-Per-Core Architecture on Application Tail Latency

Pekka Enberg
University of Helsinki

Ashwin Rao
University of Helsinki

Sasu Tarkoma
University of Helsinki

***Abstract*—The response time of an online service depends on the tail latency of a few of the applications it invokes in parallel to satisfy the requests. The individual applications are composed of one or more threads to fully utilize the available CPU cores, but this approach can incur serious overheads. The thread-per-core architecture has emerged to reduce these overheads, but it also has its challenges from thread synchronization and OS interfaces. Applications can mitigate both issues with different techniques, but their impact on application tail latency is an open question.**

We measure the impact of thread-per-core architecture on application tail latency by implementing a key-value store that uses application-level partitioning, and inter-thread messaging and compare its tail latency to Memcached which uses a traditional key-value store design. We show in an experimental evaluation that our approach reduces tail latency by up to 71% compared to baseline Memcached running on commodity hardware and Linux. However, we observe that the thread-per-core approach is held back by request steering and OS interfaces, and it could be further improved with NIC hardware offload.

***Index Terms*—Tail latency, Thread-per-core, IRQ affinity, irqbalance, Key-Value Stores.**

I. Introduction

Online services, such as e-commerce platforms and social networks, are composed of wide range of smaller applications, which communicate with each other using a request-response pattern. These services communicate with applications in parallel to reduce service-level response time. However, as a service request cannot complete until all applications have served their share of requests, the response time of the slowest application determines the overall service response time. Reducing *application tail latency* is therefore critical to ensure that service requests are completed within their latency requirements [9].

Individual applications are composed of one or more threads, and the number of threads depends on the application requirements and architecture [34], [42]. Recently, the *thread-per-core architecture* has emerged to improve throughput and reduce latency [40]. In this architecture, applications instantiate a single thread per CPU core that is available for the application. This reduces the overheads from CPU multiplexing and synchronization, and allows threads to be scheduled and balanced at coarse granularity for latency-sensitive applications [33], [36]. However, this approach is effective only when the threads can run independently.

Thread synchronization and blocking OS services are the two main issues that prevent threads from running independently. However, applications can mitigate both issues with different techniques. For example, application-level data partitioning can eliminate thread synchronization and applications can restrict themselves to using asynchronous OS interfaces. However, the need for such unconventional techniques raises a question: *what is the impact of thread-per-core architecture on application tail latency?*

To answer this question we use our key-value store that leverages application-level partitioning and inter-thread messaging, and compare its tail latency to Memcached [13], which uses a traditional multi-threaded application architecture. We also investigate the effect of interrupt processing and concurrency in multicore systems, which are reported to have an impact on tail latency [27].

We make the following contributions.

– We explore the design space of thread-per-core architecture (shared-everything, shared-nothing, and shared-something models) and the impact of interrupt handling techniques such as interrupt affinity and balancing in §III.
– We measure the impact of thread-per-core architecture on application tail latency for different combination of interrupt handling techniques in §IV.
– We discuss how thread-per-core approaches are held back by request steering overheads, OS abstractions and interfaces, and how applications could take advantage of hardware offload in §V.

II. Background

Application latency, *i.e.* the response time of a request made to an application, depends on i) the time for the request to reach a thread after it has arrived on the NIC, ii) the time for the application to process the request, and iii) the time for a response to arrive on the NIC from the thread. The components i) and iii) depend on the OS network stack, while ii) depends on thread synchronization for processing a request and creating a response. The thread-per-core approach, discussed in §III, aims to reduce application latency by addressing the issues of CPU affinity, thread synchronization, and OS interfaces.

In-kernel network stack. When a packet arrives on the NIC, the kernel device driver notices the new packet either by polling the NIC, or via an interrupt. The device driver allocates an in-kernel data structure for the packet, and forwards the packet to the in-kernel protocol stack. The protocol stack performs its processing and then notifies the application that new data is available on one of its sockets. The notification wakes up

978-1-7281-4387-3/19/$31.00

a thread which then retrieves the new data with the `recv` system call, performs its application-specific logic, and sends a response using the `send` system call. The kernel then forwards the response to to the NIC via the in-kernel network stack and the device driver.

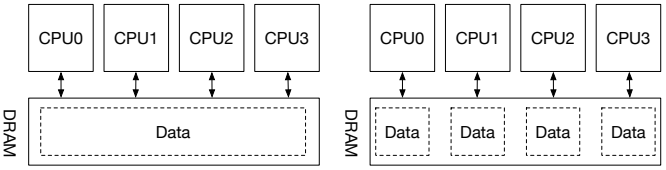
CPU affinity. On multicore servers, packets can bounce between up to three different CPU cores when traversing the in-kernel network stack. The NIC steers packet to one of its RX queues, which maps to a CPU core. When kernel software steering is enabled, the kernel then forwards the packet to another core which runs the in-kernel network stack. Finally, the kernel forwards the packet to a thread which may be running on yet another CPU core. This lack of packet processing locality decreases overall performance [19], [35]. Furthermore, a thread currently servicing a request may have to yield the CPU to the in-kernel network stack, which will process a packet that is destined for another thread that is running on a different CPU core [23].

Thread synchronization. Applications that run on multicore servers use threads for parallelism to fully utilize all the available CPU cores. However, the threads need to synchronize with each other to serve a given request if multiple threads access the same resources. This thread synchronization has two problems: (1) the synchronization itself has overheads that increases request processing latency, and (2) it limits scalability on large multicore systems. Both issues can be mitigated with various approaches including data partitioning [28] and using concurrent data structures [6], [12].

OS interfaces. Applications use the POSIX socket API to receive and transmit data over the network. The socket API uses systems calls for implementing data plane operations such as `send` and `recv`. This approach has a high overhead because of system call context switching and kernel crossing [14], [17], [19], [37], [43]. This interface also forces the kernel to copy packet data to an application buffer, amplifying the system call overhead. Furthermore, the system call overheads have recently increased because the kernel needs to mitigate against CPU security vulnerabilities such as Meltdown and Spectre [8]. Applications therefore use I/O multiplexing interfaces because the socket API does not scale to a large number of connections. On Linux, applications call the `epoll_wait` system call to wait for any of the sockets it has expressed interest in to have new data available. However, I/O multiplexing interfaces only provide notification that new data is available, but the application still needs to call the `recv` system call to obtain the data [31]. Furthermore, as packet processing can happen on CPU core which is different from the one the thread is running on, the notification can require an expensive inter-process interrupt (IPI).

III. Thread-Per-Core Architecture

In the thread-per-core application architecture, each thread is pinned to a CPU core which is dedicated for that thread. This thread is designed to fully utilize the CPU core on which it is running.



(a) Shared-everything. (b) Shared-nothing.

Fig. 1. **Example of shared-everything and shared-nothing approach**. *In the shared-everything approach, the DRAM is shared among the threads pinned to the CPU cores, while in the shared-nothing approach the DRAM in partitioned among the threads.*

A. Resource Management

An application designed using a thread-per-core approach can either (1) allow each thread to have acess to all of the underlying resources (shared-everything), or (2) partition the resources among its threads (shared-nothing), or (3) use a hybrid of the two (shared-something).

Shared-everything approach. This approach eliminates the costs of context switch between the threads while preserving the typical shared-memory architecture of many applications. However, this approach requires applications to exclude thread synchronization that block and starve the CPU cores. For instance, applications can leverage lockless data structures and use asynchronous wait-free data structures to eliminate blocking behavior.

In the example shown in Figure 1(a), the threads are pinned to the CPU cores while the data is stored in the DRAM which is shared among the threads. The threads need to synchronize when accessing the data but pinning them to the CPU cores allows them to run independently without avoiding context switches.

The main benefit of this approach is that it can maximize system throughput because any CPU core can be used to serve the requests. The problem in the shared-everything approach is that data bounces between CPU caches and that thread synchronization limits multicore scalability [16].

Shared-nothing approach. Each thread only accesses the resources assigned to it. This eliminates the need for locks because each thread is independent of the other threads. While this requires threads to communicate with each other, Barrelfish [5] and Seaster [40] have exemplified scenarios where partitioning and message passing are less expensive than shared memory and locking.

In the example shown in Figure 1(b), each thread is allocated a portion of the data stored in the DRAM. This thread is now responsible for responding to the queries related to the data allocated to it. However, when a thread receives a request for the data it is not responsible for, then it must forward the request to the appropriate thread.

The main advantage of this approach is that improves CPU cache efficiency and eliminates thread synchronization. How-

ever, this approach can limit system throughput for skewed workloads because only one CPU core can operate on a specific part of the application data.

Shared-something approach. Application still need to partition some resources, but more than one CPU core can access the same data. The shared-something approach complements the sub-NUMA clustering approach of Intel's Skylake microarchitecture, which partitions CPU cores around memory controllers within a NUMA domain [32].

B. Interrupt Affinity

The network stack performs the protocol processing in kernel threads serving software interrupts (softirqs). When the system is idle, the NIC generates an interrupt request (IRQ) on arriving packets. The interrupt handler forwards the packets to a kernel thread which performs protocol processing and sends them to the application threads. If the arrival rate of packets is high, the network stack starts polling the NIC.

On Linux, the irqbalance daemon distributes the IRQs over the available CPU cores [1]. Li *et al.* [27] recommend running interrupts on dedicated cores to avoid the irqbalance daemon spreading interrupts to cores that are running application threads. This can be achieved by leveraging IRQ affinity which allows us to specify which CPUs can serve a given IRQ [2], *i.e.* IRQ affinity can be used to specify the CPU cores on which the in-kernel network stack runs. Running the network stack on some dedicated CPU cores improves performance because of (1) more efficient packet processing and (2) reduced thread disturbance [21], [41]. Clearly, it is desirable that IRQ are served on the CPU cores on which the application threads are not running. However, the impact of IRQ affinity and IRQ balancing on the performance of the thread-per-core architecture is largely unknown.

IV. Impact on Tail Latency

We use the following two key-value stores (KV stores) for quantifying the impact of the thread-per-core application architecture: Memcached [13], and our custom key value store named Sphinx. We compare the performance of Sphinx against Memcached because it has a threaded, shared-memory architecture, and it is the state-of-the-practice. Furthermore, Memcached emulates a shared-everything thread-per-core architecture when its thread pool size is configured to match the number of CPU cores. We also considered comparison to MemC3 [12], but ruled it out because we experienced server crashes under load in our experimental setup. We decided against comparing to solutions that require OS modifications (*e.g.*, DPDK) or specific hardware (*e.g.*, FPGA).

A. Application Architecture

Sphinx's architecture, shown in Figure 2, is designed around two principles: (1) partitioning of OS resources and application data between the CPU cores, and (2) message passing between the threads. Its event loop leverages asynchronous OS services (*i.e.*, non-blocking socket system calls) and I/O multiplexing (*i.e.*, epoll) to implement a thread-per-core model. Sphinx is not a replacement for Memcached and it implements only a subset of the Memcached wire protocol. However, Sphinx is compatible enough with Memcached clients to allow us quantify the impact of the thread-per-core architecture.

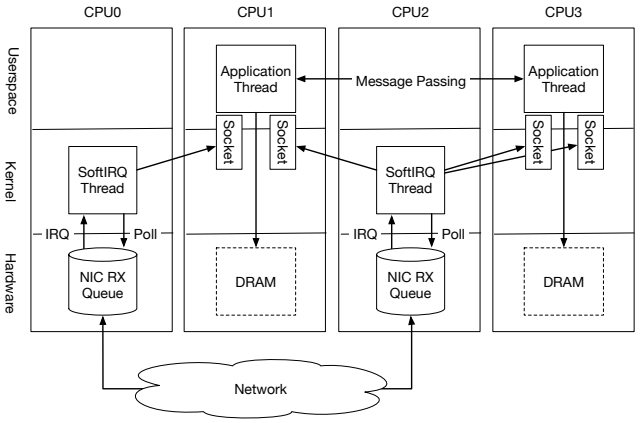


Fig. 2. **Architecture overview.** *Sphinx leverages application-level partitioning and message passing to improve request-level parallelism and improve single-thread performance on commodity hardware and Linux.*

Application-level partitioning. Sphinx leverages partitioning, similar to MICA [28], but unlike MICA it extends the architecture to commodity OS and hardware by partitioning OS resources and application data between its threads. MICA uses kernel-bypass networking to direct packets arriving on a NIC to the core that manages the key present in the request. MICA also requires the clients to be aware of the partitioning scheme for the data and it works only with stateless protocols such as UDP. In contrast, Sphinx also partitions OS resources such as sockets and leverages user space message passing to forward requests between cores. When Sphinx starts up, it spawns a thread on each of the CPU cores for which it is configured to run on. Sphinx assigns each key to a specific core, and the thread running on a core is responsible for storing the data on the partition of the DRAM allocated to that core. The keys are partitioned uniformly between cores by hashing the key with MurmurHash3 and taking the modulo of the number of threads. Sphinx also partitions sockets between cores to avoid OS internal lock contention from multiple threads accessing the same socket. Furthermore, by using the `SO_REUSEPORT` socket option, Sphinx lets the network stack distribute connections across the threads to ensure full utilization of all cores [22], [30]. Note that NIC IRQ processing can run on dedicated cores or on cores running Sphinx threads, depending on Sphinx's configuration.

Inter-thread messaging. When a thread receives a request via a connection it manages, it first checks if it manages the key present in the request. If it manages the key, it performs the requested operation locally and sends a response. However, if another thread manages the key, it uses message passing to forward the request to that remote thread. The remote thread performs the request operation and sends back a message

TABLE I
HARDWARE CONFIGURATION

Modern hardware	
CPU	Intel Xeon CPU E5-2680 v3 @ 2.5 0GHz
# Cores	2 x 12 cores (48 hardware threads)
L1/L2/L3	768 KiB, 3 MiB, 30 MiB
Memory	256 GiB DDR4 @ 2133 MT/s
NIC	Intel 82599ES 10-Gigabit SFI/SFP+
# Queues	48 RX queues, 48 TX queues
Legacy hardware	
CPU	Intel Xeon E5540 CPUs @ 2.53 GHz
# Cores	2 x 4 cores (16 hardware threads)
L1d/L1i/L2/L3	32 KiB, 32 KiB, 256 KiB, 8192 KiB
Memory	32 GiB DDR3 @ 1066 MT/s
NIC	Broadcom NetXtreme II BCM57710
# Queues	8 RX queues, 24 TX queues

to the original thread which responds to the request. The purpose of this design is to ensure accesses to sockets are local to a specific core, which eliminates contention on Linux networking stack locks. Sphinx implements message passing in userspace with shared memory and system calls for waking up a receiver thread to eliminate IPC overheads. The threads communicate with each other using a bounded, lock-free, and single-producer single-consumer (SPSC) queue [29] which eliminates the need for cache line sharing between producers. Furthermore, we chose eventfd over signals as the wakeup mechanism because the latter acquires a process-wide signal handler spinlock.

Data structures. Sphinx uses the C++ standard library `unordered_map` as the data structure for key-value index. The key and value in the index are references to a memory region, which are private to a thread that is responsible for a given key. A log-structured memory allocator (LSMA) [38], which stores both keys and values, manages the thread memory region. LSMA divides the thread memory region into fixed-size segments, which are the size of a large page to reduce TLB pressure (*i.e.*, 2 MiB on x86). The allocator appends data to a segment until it runs out of free space and then picks a new segment until all segments are full. If the allocator needs to reclaim memory to accommodate a new request, it expires whole segments in FIFO order. Sphinx does not compact segments because of the cache-like semantics of the Memcached protocol that specifies that new operations may expire old data.

B. Evaluation Setup

We selected two different multicore configurations for the evaluation, modern and legacy, which are detailed in Table I. In the modern configuration, which represents a high-end instance on the cloud, we use a server with 48 logical cores and a multi-queue NIC that has RX and TX queues for every logical core. In the legacy server configuration, which represents a medium-end cloud instance, we use a server with 16 logical cores and a multi-queue NIC that has 8 RX and 24 TX queues. The number of RX queues is less than the number of logical cores in the legacy configuration.

To answer the question of the impact of NIC IRQ isolation on tail latency, we focus on IRQ balancing and IRQ affinity and repeat the experimental evaluation for the following configurations.

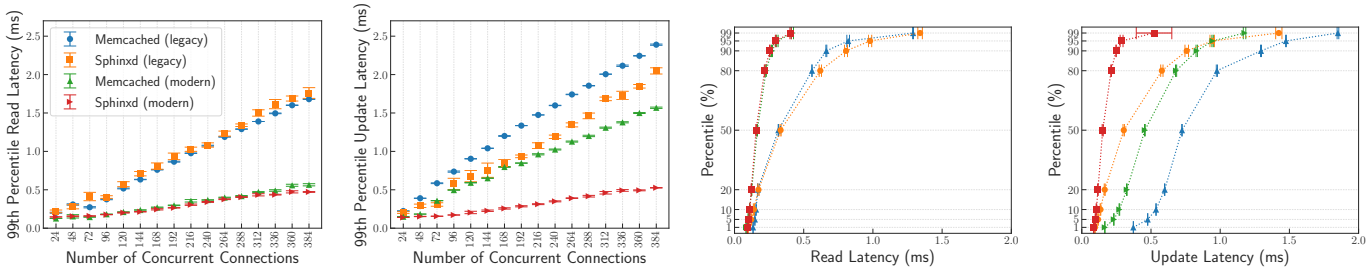
– *IRQ affinity not configured, and IRQ balance enabled:* The irqbalance is enabled and no IRQ affinity is configured. We use this configuration as the baseline because it the default configuration on most systems.
– *IRQ affinity not configured, and IRQ balance disabled:* The irqbalance is disabled and no IRQ affinity is configured. This configuration isolates the impact of the irqbalance on tail latency.
– *IRQ affinity configured, and IRQ balance disabled:* IRQ affinity is configured so that NIC IRQs are handled by dedicated cores and remaining cores are running threads. The irqbalance daemon is disabled to eliminate interference with the IRQ affinity configuration. This configuration is expected to give the lowest latency [27], [41].

We do not consider a configuration with IRQ affinity configured with the irqbalance daemon enabled because the irqbalance daemon currently overrides IRQ affinity, and the current implementation of this override further limits the performance [18].

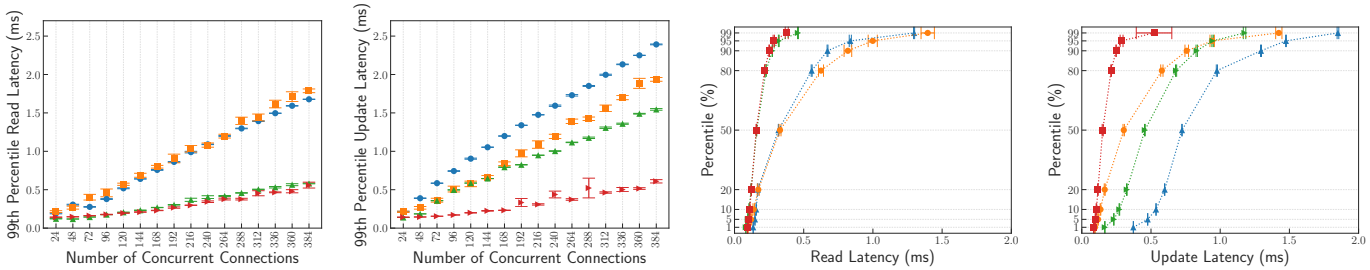
We compare the latency performance of Memcached and Sphinx KV store in the following manner. We use two servers at time: one server is used to run Mutilate [25], our load generator, and the other to run the KV store being evaluated. Each KV store is successively run on a modern server followed by the legacy server, while Mutilate ran on a modern server. We configured Mutilate to use 24 worker threads to avoid overloading the server on which it was running. Mutilate is configured to use default key and value sizes, 30 and 200 bytes, respectively. The set/get ratio of Mutilate is configured as 0.0 for read-only workload and 1.0 for update-only workload. Every test is run for 50 seconds and repeated 5 times. When IRQ affinity is not configured, Memcached and Sphinx are configured to use one thread per hardware thread (48 threads on modern hardware and 16 threads on legacy hardware). When IRQ affinity is configured, Memcached and Sphinx are configured to threads on CPUs that are not handling NIC IRQs. Specifically, we allocate 8 CPUs for serving NIC IRQs and 40 CPUs for running threads on modern hardware, and we allocate 4 CPUs for serving NIC IRQs and 12 CPUs for running threads on legacy hardware. We vary the number of concurrent connections per Mutilate worker thread from 1 to 16, resulting in a total number of concurrent connections between 24 and 384.

C. Tail Latency Results

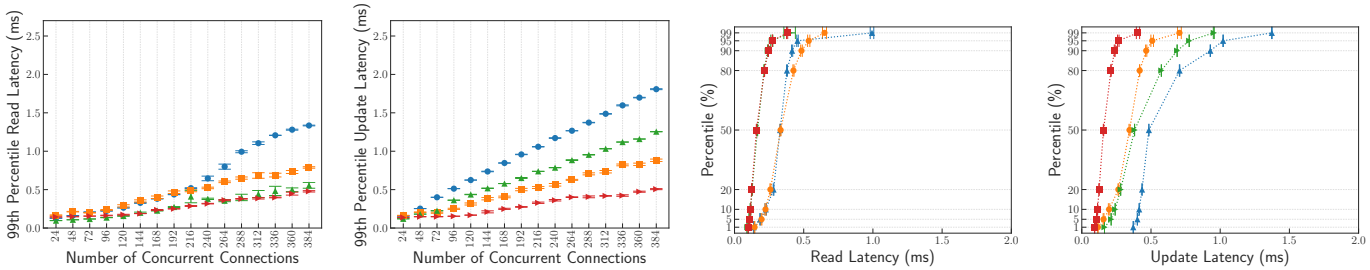
We observe that running Memcached and Sphinx with IRQ affinity not configured and IRQ balance enabled (Figure 3(a)) exhibits a high tail latency for read and also update operations. In contrast, we observe the lowest tail latency for the read and the update operations when IRQ affinity is configured and IRQ balance is disabled (Figure 3(c)). This is inline with the recommendation of Li *et al.* [27].



(a) IRQ affinity not configured, and IRQ balance enabled.



(b) IRQ affinity not configured, and IRQ balance disabled.



(c) IRQ affinity configured, and IRQ balance disabled.

Fig. 3. **Latency results.** *In each subfigure, we present from left to right i) the the 99th percentile read latency for a given number of concurrent connections, ii) the 99th percentile update latency for a given number of concurrent connections, iii) the percentiles for the read latency for 288 concurrent connections, and iv) the percentiles for the update latency percentiles for 288 concurrent connections. When IRQ affinity is configured and IRQ balance is disabled, i.e., the configuration recommended to run Memcached [27], we observe that Sphinx exhibits a lower tail latency for read and update operations compared to Memcached.*

Across all configurations we observe that the tail latency for the read and update operations increases with the number of concurrent connections. We observe that the tail latency of Sphinx for the update operations is lower than the corresponding values of Memcached. In contrast, we observe that the tail latency of Sphinx for the read operations is lower than Memcached only when a) IRQ affinity is enabled and IRQ balance is disabled, and b) for a large number of concurrent connections. This shows that *Sphinx is capable of exploiting the CPU resources to reduce the tail latency*.

In Figure 3, we observe that *IRQ balance has a smaller impact on the read and update latency of Sphinx and Memcached compared to IRQ affinity*. One reason for this behavior is that when IRQ affinity is not configured, the CPU on which the threads are running need to also serve the interrupts arriving at the NIC RX queue.

When IRQ affinity is not configured and IRQ balance is enabled (the baseline configuration), Sphinx has up to 16% lower read tail latency than Memcached on modern hardware. On legacy hardware, Sphinx's read latency is up to 9% lower than that of Memcached. Similarly, Sphinx's update tail latency is up to 67% and 47% lower than that of Memcached on modern and legacy hardware, respectively. These results highlight the effectiveness of Sphinx's architecture in the eliminating overheads of user space locking. However, Sphinx's read and update tail latency are higher than Memcached's for a small number of concurrent connections. *This shows a weakness of the thread-per-core model for workloads that do not distribute work to all cores.*

When IRQ affinity is not configured and IRQ balance is disabled (Figure 3(b)), Sphinx's read and update tail latency are unaffected compared to baseline. However, Memcached's read tail latency is up to 23% lower than baseline on modern hardware but just 4% lower on legacy hardware. Memcached's update latency is 7% and 4% lower than baseline on modern and legacy hardware, respectively. *Disabling the irqbalance daemon reduces Memcached's tail latency and has no impact on Sphinx's tail latency, which makes it a robust optimization to reduce tail latency.*

Finally, when IRQ affinity is configured and IRQ balance is disabled (Figure 3(c)), Sphinx's read latency is up to 20% lower than Memcached's on modern hardware and up to 54%

lower on legacy hardware. Furthermore, Sphinx's update tail latency is up to 71% and 66% lower than baseline for modern and legacy hardware, respectively, which highlights that *IRQ affinity is very effective for Sphinx's thread-per-core model.*

V. Discussion

Our measurements show that the thread-per-core architecture combined with data partitioning is capable of reducing the tail latency. At the same time, building and testing Sphinx gave us the following insights on the factors which are holding back its performance.

Request steering. Request steering can improve tail latency, for example, by prioritizing small requests over large ones to eliminate head-of-line blocking [10]. However, software-based request steering, such as message passing between threads used by Seastar and our prototype KV store, suffer from high overheads because of thread wake-ups [24]. The lack of OS interfaces to perform the wake-up forces applications to use POSIX signals or OS-specific interfaces, such as `eventfd`, to perform the wake-up. Hardware steering approaches supported by NICs are currently limited to flow-based steering. For example, the MICA assigns a separate UDP port to each CPU core and uses the NIC Flow Director to steer requests. The problem with this approach is that it requires the client to specify the correct UDP port, thus making the partitioning scheme visible to the client. We therefore plan to look at using programmable NICs for performing application-level packet steering.

OS interfaces and abstractions. Kernel-bypass interfaces eliminate many of the in-kernel network stack overheads. However, the lack of standard OS interfaces is holding back their adoption. Recently, Linux has introduced Express Data Path (XDP) interface, which provides an in-kernel packet processor and a kernel-bypass interface. Similarly, Demikernels propose OS interfaces that apply to a wide range of kernel-bypass accelerators [44] while parakernels propose partitioning as first-class OS principle [11].

Hardware offload. CPU speeds have stagnated [15], [39] and they are falling behind NIC speeds [20]. Hardware offload approaches are therefore needed to keep up with the fast rate of packets arriving from the network. The KV-Direct KV store [26], for example, offloads whole application to an FGPA running on the NIC. This approach yields massive performance gains because computation moves to the network. However, offloading complete applications to a FGPA is complex and expensive. One research direction is to look into how applications can take advantage CPU and NIC offload co-design, where only some parts of the application is offloaded.

Other approaches to application-level parallelism. Arachne is a core-aware, user-level threading library that aims to provide high throughput and low latency for short-lived threads (in the order of few microseconds) [36]. Arachne is an example of a thread-per-core, shared-everything model. By replacing Memcached's thread-pool with Arachne, they demonstrate the benefits of running a single kernel thread per CPU core, and pinning it to avoid inference from other threads.However, Arachne does not solve the issue of blocking kernel operations (for example, system calls or page faults), and leaves thread synchronization to users of the threading library, stating that applications should use asynchronous interfaces and avoid paging.

Shenango makes the observation that state-of-art systems waste CPU cycles to achieve microsecond-scale tail latency by busy-polling the NIC to detect arriving packets [33]. To address this inefficiency, Shenango dedicates a single CPU core for running an IOKernel, which busy-polls the NIC for packets, steers them to applications, and reallocates CPU cores as per application run-time needs.

Future research directions. In this paper, we focus on how to apply the thread-per-core architecture efficiently to address the overheads of kernel threads and thread synchronization. We implemented a prototype key-value store based on this architecture, Sphinx, which used application-level partitioning, inter-thread messaging, and IRQ isolation. We then used small request sizes for studying the impact of application-level partitioning and IRQ isolation. We envision to extend this work by considering different workloads such as YCSB [7], and the *Facebook "ETC" request stream* [4], [25] for emulating large value sizes which are prevalent [3]. These workloads will also help us evaluate the throughput and study the impact of message passing. Initial evaluation results from YCSB indicate that our current approach for message passing is inefficient for larger request sizes because packet steering involves a memory copy. We think that the issue is best addressed by co-designing the application-level partitioning scheme with NIC hardware offload using programmable NICs. Furthermore, we believe that addressing the current OS interface limitations that are holding back the efficiency of the thread-per-core model is critical.

VI. Conclusion

Application tail latency is critical for services to meet their latency expectations. We have shown that the thread-per-core approach can reduce application tail latency of a key-value store by up to 71% compared to baseline Memcached running on commodity hardware and Linux. However, we believe that the thread-per-core approach is held back by request steering and OS interfaces, and it could be further improved with CPU and NIC offload co-design.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback, and Lirim Osmani for helping us setup the experimental evaluation testbed.

Availability

The source code of Sphinx is available at https://github.com/penberg/sphinx in branch `ancs19`.

REFERENCES

[1] "Irqbalance," irqbalance.github.io/irqbalance/, [Online; accessed 2019-07-22].

[2] "SMP IRQ affinity," https://www.kernel.org/doc/Documentation/IRQ-affinity.txt, [Online; accessed 2019-07-18].

[3] A. Adya, R. Grandl, D. Myers, and H. Qin, "Fast Key-value Stores: An Idea Whose Time Has Come and Gone," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: ACM, 2019, pp. 113–119. [Online]. Available: http://doi.acm.org/10.1145/3317550.3321434

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254766

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579

[6] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "FASTER: A Concurrent Key-Value Store with In-Place Updates," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 275–290. [Online]. Available: http://doi.acm.org/10.1145/3183713.3196898

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[8] J. Corbet, "The impact of page-table isolation on I/O performance," https://lwn.net/Articles/752587/, [Online; accessed 2019-07-22].

[9] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794

[10] D. Didona and W. Zwaenepoel, "Size-aware Sharding for Improving Tail Latencies in In-memory Key-value Stores," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. Berkeley, CA, USA: USENIX Association, 2019, pp. 79–93. [Online]. Available: http://dl.acm.org/citation.cfm?id=3323234.3323242

[11] P. Enberg, A. Rao, and S. Tarkoma, "I/O Is Faster Than the CPU: Let's Partition Resources and Eliminate (Most) OS Abstractions," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: ACM, 2019, pp. 81–87. [Online]. Available: http://doi.acm.org/10.1145/3317550.3321426

[12] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 371–384. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482662

[13] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, vol. 2004, no. 124, Aug. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1012889.1012894

[14] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 135–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387894

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[16] D. A. Holland and M. I. Seltzer, "Multicore OSes: Looking Forward from 1991, Er, 2011," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 33–33. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991640

[17] T. Hruby, T. Crivat, H. Bos, and A. S. Tanenbaum, "On Sockets and System Calls: Minimizing Context Switches for the Socket API," in *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*, ser. TRIOS'14. Berkeley, CA, USA: USENIX Association, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2750315.2750323

[18] Irqbalance, "Remove affinity_hint infrastructure," https://github.com/Irqbalance/irqbalance/commit/dcc411e7bfdd95bbdb7fd0af8699f8cafe686ff4, [Online; accessed 2019-07-22].

[19] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *11th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI 14. Seattle, WA: USENIX Association, 2014, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[20] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 67–81. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872367

[21] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, "TAS: TCP Acceleration As an OS Service," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: ACM, 2019, pp. 24:1–24:16. [Online]. Available: http://doi.acm.org/10.1145/3302424.3303985

[22] M. Kerrisk, "The SO_REUSEPORT socket option," https://lwn.net/Articles/542629/, [Online; accessed 2019-07-22].

[23] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based CPU in container environments," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 313–328. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/khalid

[24] A. Kivity, "Adventures with Memory Barriers and Seastar on Linux," https://www.scylladb.com/2018/02/15/memory-barriers-seastar-linux/, [Online; accessed 2019-07-22].

[25] J. Leverich and C. Kozyrakis, "Reconciling High Server Utilization and Sub-millisecond Quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 4:1–4:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592821

[26] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 137–152. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132756

[27] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 9:1–9:14. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670988

[28] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-memory Key-value Storage," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 429–444. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616488

[29] N. M. Lê, A. Guatto, A. Cohen, and A. Pop, "Correct and Efficient Bounded FIFO Queues," in *2013 25th International Symposium on Computer Architecture and High Performance Computing*, Oct 2013, pp. 144–151.

[30] M. Majkowski, "Why does one NGINX worker take all the load?" https://blog.cloudflare.com/the-sad-state-of-linux-socket-balancing/, [Online; accessed 2019-07-22].

[31] S. Marz and B. V. Zanden, "Reducing Power Consumption and Latency in Mobile Devices Using an Event Stream Model," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 1, pp. 11:1–11:24, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2964203

[32] D. Mulnix, "Intel Xeon Processor Scalable Family Technical Overview," https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, 2017, [Online; accessed 2019-07-22].

[33] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. Berkeley, CA, USA: USENIX Association, 2019, pp. 361–377. [Online]. Available: http://dl.acm.org/citation.cfm?id=3323234.3323265

[34] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable web server," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268708.1268723

[35] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving Network Connection Locality on Multicore Systems," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 337–350. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168870

[36] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-Aware Thread Management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/qin

[37] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[38] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured Memory for DRAM-based Storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2591305.2591307

[39] K. Rupp, "42 Years of Microprocessor Trend Data," https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, [Online; accessed 2018-02-25].

[40] Seastar, http://www.seastar-project.org/, [Online; accessed 2019-07-22].

[41] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda, "IsoStack: Highly Efficient Network Processing on Dedicated Cores," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855845

[42] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-conditioned, Scalable Internet Services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 230–243. [Online]. Available: http://doi.acm.org/10.1145/502034.502057

[43] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 43–56. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[44] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, "I'm not dead yet!: The role of the operating system in a kernel-bypass era," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: ACM, 2019, pp. 73–80. [Online]. Available: http://doi.acm.org/10.1145/3317550.3321422