Performance Exploration of NIC Features in a Baremetal Library OS

Abstract

We identify four hardware features in the Intel 82599 10 GbE NIC that can speed up packet processing and conducted an experimental study where these features are explored for improving the performance of a memcached workload. We use a baremetal library OS as a platform to conduct these experiments to demonstrate a 1.71X performance improvement by selectively enabling these hardware features. We also collect hardware and software statistics to better understand their performance impacts. Having established potential value in using these features, we then enabled three of the feature in Linux via small changes. While this did not achieve in performance improvements, our overall results suggest their applicability in other systems built to accelerate memcached with more aggressive systemic changes.

1 Introduction

Modern hardware is complex [15, 21, 22, 25] with many features packed into it, for example, we reviewed the programming datasheet for the Intel 82599 family of 10 GbE NICs [11] and counted a total of 5630 hardware registers (each 32-bit) that can be configured. While the vast majority of these registers deal with correctly setting up the NIC, we identified four features (details in section 2) that impact the efficiency of packet processing in the NIC. To gain a better understanding of these features and their potential benefits, we conducted an experimental study in toggling these features for Memcached [7] while running in a library OS, EbbRT [24]. We use memcached as an example application as key-value stores are widely deployed by many companies [1, 17, 20, 23, 26].

In order to isolate and focus on the NIC features, we were motivated to use EbbRT over Linux due to its smaller codebase and ease of modification. EbbRT is open-sourced and is able to boot baremetal, furthermore, it provides a device driver written in C++ to support the Intel 82599 NIC. As a configurable library OS, EbbRT consists of 15K lines of code with an additional 4K lines of code in its NIC driver. For this paper, we extended its NIC driver to enable the four hardware features and instrumented additional data collection for both hardware and software statistics. Our approach helps to experimentally validate the advantages of these features without requiring systemic modifications that impact various subsystems, potentially perturbing experimental conclusions. In our experimental results, we find performance improvements in memcached by up to 1.71X through the selective use of these hardware features **without modification to application or system code**. Furthermore, we instrument data collection of hardware and software statistics in the device driver to better understand how these features can impact performance. Our result also suggest previous work in optimizing memcached [3, 5, 8, 9, 13, 16, 18, 19, 27, 28] can benefit from using these hardware features as they share similar qualities as EbbRT such as kernel-bypass, no domain-crossings, run-to-completion of every request, etc.

Next, given the performance improvements from the EbbRT experiments, we then enabled three of these features in Linux with small changes. However, while this did not result performance improvements in Linux memcached, we document our methods to enable these features and how these features can interact with other Linux subsystems. Linux's result also suggest the difficulty in taking advantage of these features in a general purpose setting due to the overall complexity of OS work that still needs to be done per request after packets have been processed by the NIC.

The rest of the paper is structured as follows: section 2 provides greater detail of the four hardware features, section 3, 4 details experimental setup and results, section 5 summarizes our attempts in Linux and the paper concludes in section 6.

2 82599 NIC Features Explained

In order to put the NIC features explored in this paper into context, we first provide an overview of the process with which packets are transmitted and received through the coordination between device driver and NIC. This coordination is achieved

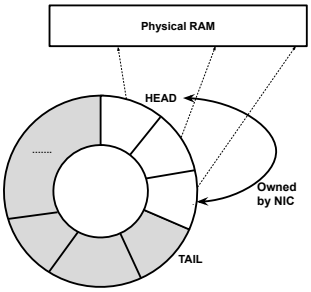

Figure 1: Illustration of the NIC's descriptor ring and its behaviors.

via per-core descriptor rings, one each for packet transmissions and receives. An illustration of a ring is provided in fig. 1, where each chunk in the ring is a single descriptor. A descriptor is a data structure that provides information to the NIC about the packet to be processed (such as length, checksums, etc) as well as referencing a location in main memory in order for the NIC to transmit or receive packet data. In legacy NICs, each descriptor typically referenced MTU sized data blocks to be processed. Using packet transmissions as an example, this required the application to manually fragment the original data first. However, the evolution of systems software and hardware has moved most of this work to the network stack and to the NIC.

Fig. 1 also shows the use of *HEAD* and *TAIL* pointer variables to delineate ownership of descriptors between the device driver and the NIC. During normal operation, the NIC will process packet descriptors pointed to by *HEAD* until *TAIL*; whereas the device driver will add work to the NIC by incrementing *TAIL* and notifying the NIC about the new work. Using fig. 1 as an exemplar, the following subsections will now explain how the NIC features explored in this paper expedites packet processing.

2.1 Transmit Head Pointer Write-back (TWB)

After every successful transmission by the 82599 NIC, it will mark a bit inside the transmit descriptor to let the device driver know it can now reclaim the memory for reuse. Normally, the device driver needs to periodically check this bit in order to ensure packet transmission has completed and then increment *HEAD* pointer, however, accessing this complex data structure by both the device driver and NIC can be expensive due to potential cache thrashing. By enabling TWB, the NIC can now automatically increment *HEAD* pointer instead.

We did not find this feature enabled in Linux's 82599 device driver. We suspect this could be partially due to NAPI budgets that enforce a limit for how many transmit descriptors can be processed within some quantum; TWB allows the NIC to exceed that budget. We toggle this feature on and off in EbbRT's device driver to study its performance implications.

2.2 Transmit Segmentation Offload (TSO)

TSO is a common feature that is supported by Linux and allows the device driver to give the NIC a large payload so that the NIC can automatically break the payload into multiple MTU sized frames with the appropriate Ethernet, TCP/IP header information. This is achieved by simply using multiple descriptors to point to different chunks of the payload. There are two hardware limits as defined by the 82599 NIC: 1) a single TSO payload must be under 256 KB due to memory limitation of the NIC, and 2) only up to 40 transmit descriptors may be used per TSO payload. To explore impacts of TSO, we modified EbbRT's 82599 device driver to investigate TSO in two manners: 1) always use a **single** descriptor by copying payload to a single buffer, and 2) use **multiple** descriptors (up to max of 40). The motivation behind this is to better understand performance extremities of this feature.

Similar to TWB, TSO use is limited in Linux as 1) NAPI budgets place a limit on how aggressive the device driver can take advantage of multiple descriptors, and 2) data structures such as socket buffers already do some of the truncation in software.

2.3 Direct Cache Access (DCA)

DCA is a mechanism where incoming packet data can be written directly into CPU caches via a hardware pre-fetch. Enabling DCA when the system is under load can cause potential cache line evictions that can negatively impact the application and previous work in studying DCA [4,6,10] have revealed its performance benefits and unpredictable-ness. This paper is the first to present results with other NIC features together and in a baremetal Library OS.

2.4 Receive Side Coalescing (RSC)

RSC is the NIC's implementation of the large receive offload [14] in hardware. On reception of a payload spanning multiple descriptors for a single flow, the NIC can automatically merge the packet headers and data such that the software only needs to process a single packet header with the rest of the payload containing the data. This feature helps move the cost of some header processing into the NIC itself. RSC is typically disabled by default in Linux as it drops potential header information used in TCP/IP routing and bridging.

3 Experimental Setup

In order to investigate the interactions of the four hardware features on a commonly deployed server workload such as a memcached server, we use the features by toggling them on/off in different combinations with the goal of optimizing the server's requests-per-second (QPS) while maintaining a SLA of 99% tail latency under 500 us. Our experimental

setup consists of six nodes, each having 16-core processors of either Intel(R) Xeon(R) CPU E5-2690 @2.90GHz or Intel(R) Xeon(R) CPU E5-2650 @2.60GHz type. All processors have Intel 82599ES 10-Gigabit SFI/SFP+ NICs, and are configured with a mix of 126 GB and 250 GB RAM.

To benchmark the memcached server, we use an unloaded client node to run mutilate [12] as the workload generator. This client (1) coordinates with four other mutilate agent nodes in order to generate the actual requests to the server and (2) measures tail latency of all requests made. All four agent nodes consist of 16-core machines, whereby each core creates 16 connections, for a total of 1024 connections. This setup is able to saturate the single 16-core server. We run a representative load from Facebook [2] (ETC) which represents the highest capacity deployment. It uses 20 to 70 byte keys and 1 byte to 1 KB values and contains 75% GET requests. The machine used to boot into baremetal EbbRT memcached server uses a 16-core Intel(R) Xeon(R) CPU E5-2690 @2.90GHz processor with 126 GB of RAM.

Each experiment consists of generating a fixed QPS load in chunks of 200K for 10 seconds each until a peak QPS is reached where the SLA is violated. [1]. We repeated each experiment 5 times for stability and the standard deviation for the statistics collected is around 0.01%. For each chunk of experiment, two pieces of data are collected: 1) the measured 99% tail latency at that QPS, and 2) a set of hardware and software statistics that have been instrumented inside EbbRT's device driver. To better understand the impact of these hardware features on performance, we used Intel's performance monitoring counters (PMCs) to measure CPU instructions, cycles, and last-level cache (LLC) misses, and software statistics such as number of transmit and received bytes and descriptors processed.

4 Experimental Analysis

In table 1, the overall peak QPSes achieved in EbbRT is shown. The results are separated between TSO single and multiple. For each TSO, the combinations of DCA, RSC, and TWB is toggled and shown. For the rest of the analysis, this paper will focus on better understanding results of each hardware feature.

4.1 Transmit Segmentation Offload (TSO)

Surprisingly, table 1 shows that just using a single descriptor for memcached (DCA, RSC, TWB disabled) resulted in a 1.57X performance difference improvement over using multiple descriptors. The main difference between the two versions is that TSO single will suffer an extra memory copy cost before transmission to only use a single descriptor. EbbRT provides the `IOBuff` primitive for managing data with hardware devices. Zero-copy of all network data is supported by wrapping both received and transmitted payloads into `IOBuffs` for application processing. Chaining of multiple `IOBuff` is also used to further eliminate data copying. In the device driver, each `IOBuff` in this chain is mapped to a new transmit descriptor for TSO multiple.

TSO	DCA	RSC	TWB	Peak QPS (K)
single	✗	✗	✗	2200
	✗	✗	✓	2300
	✗	✓	✗	2400
	✗	✓	✓	2400
	✓	✗	✗	2300
	✓	✗	✓	2400
	✓	✓	✗	2350
	✓	✓	✓	2400
multiple	✗	✗	✗	1400
	✗	✗	✓	1000
	✗	✓	✗	1400
	✗	✓	✓	1200
	✓	✗	✗	1200
	✓	✗	✓	1200
	✓	✓	✗	1400
	✓	✓	✓	1200

Table 1: Measured performance of each NIC feature configuration when running memcached. TSO is separated into *single* and *multiple* categories (details in section 2.2), and for the rest of the hardware features (DCA, RSC, TWB), this table lists the peak QPS achieved for every combination while maintaining SLA.

To better understand these differences, fig. 2 shows the collected hardware and software statistics between TSO single and multiple at a fixed QPS of 1000K as it was still a relatively light load for both configurations. In this figure, one can see that though TSO single suffered higher LLC misses due to the extra memory copies, its instruction usage only increased slightly. This is partially due to use of EbbRT's `SlabAllocators` design that is more efficient for small memory footprints as it can allocate from identity mapped virtual memory; this is also impacted by the ETC workload which consists of small payloads less than 1 KB. The use of transmit descriptors is most evident in fig. 2 as TSO multiple used 50% more descriptors than TSO single for the same workload. To understand this better, fig. 3 shows a histogram breakdown of the number of descriptors per packet transmission. For TSO multiple we found up that to 20 descriptors were actually used per transmission, however, the vast majority was concentrated in only using 2/3 descriptors as rest were too small to be legible in the figure. Given this, we hypothesize that the TSO feature reveals a efficiency difference between the CPU and the NIC. Namely, the logic on the NIC to process each transmit descriptor and then invoke the necessary DMAs to

[1] Due to experimental complexity, we only conducted experiments at 50K granularity when closer to peak QPS to expose smaller performance differences.

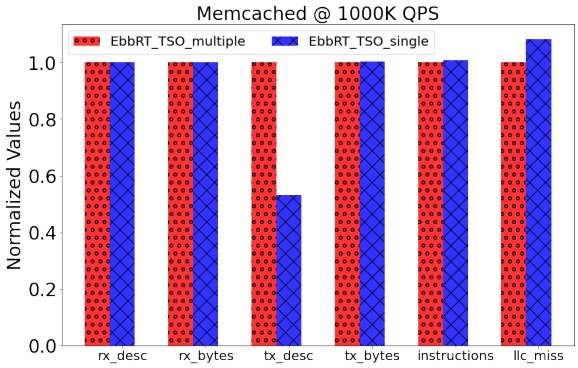

Figure 2: Collected statistics normalized against **EbbRT_TSO_multiple** for memcached at a fixed QPS load of 1000K. [rx/tx]_desc = total Receive/Transmit Descriptors used. [rx/tx]_bytes = total Receive/Transmit Bytes used.

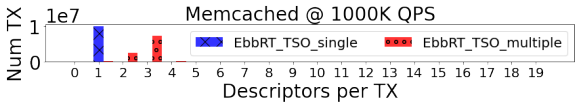

Figure 3: Num TX is a count of how many transmissions a particular descriptor count (Descriptors per TX) was used - e.g. if Descriptors per TX column was 3 and Num TX was 1000, then 1000 packet transmissions used 3 descriptors per transmission.

retrieve payloads for transmissions is not keeping pace with CPU packet processing.

To help understand the observations above, we used mutilate to generate a heavier load on both configurations using fixed 400B keys and 8KB values in order to increase load onto CPU and memory. With a heavier load, we found that both configurations converged to roughly the same peak QPS at around **170K**. Fig. 4 shows a breakdown of collected statistics at 170K QPS and one can see that the larger LLC miss difference and memory copy costs for TSO single (compared to fig. 2) compensates for the inefficiencies in using multiple descriptors.

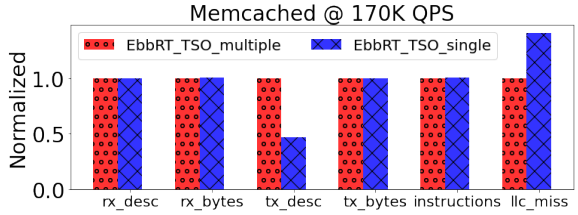

Figure 4: Using a custom large payload of 400B keys and 8 KB values at peak QPS of 170K.

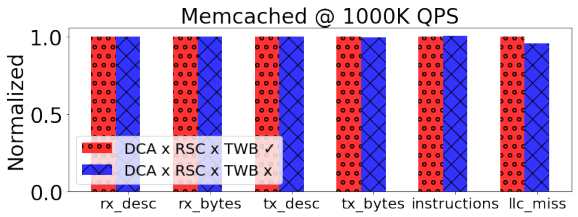

Figure 5: Collected statistics for TSO multiple and normalized against TWB enable.

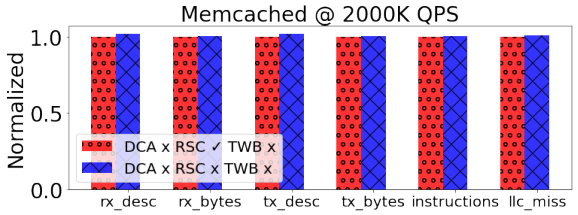

Figure 6: Collected statistics for TSO single and normalized against RSC enable.

4.2 Transmit Head Pointer Write-back (TWB)

In table 1, one can see two opposite effects with TWB enabled. For TSO single, it increased performance by 4.5% and in TSO multiple, it actually lowered performance by 29%. To better understand this behavior, we examined the collected statistics at various QPS loads (1000K is shown in fig. 5). However, we did not see any particular data point that really explained this behavior. We hypothesize this performance degradation with TSO multiple when TWB is enabled may be due to the speed at which the NIC can increment the *HEAD* pointer as TSO multiple uses many transmit descriptors and therefore relies on the NIC update more frequently in contrast to the case of TSO single.

4.3 Receive Side Coalescing (RSC)

Table 1 shows that RSC resulted in a 9% performance improvement for TSO single and did not negatively affect performance in TSO multiple. For TSO single, we show the collected statistics at 2000K QPS when RSC is enabled and disabled in fig. 6. Interestingly, this figure shows that enabling RSC resulted in a roughly 2% decrease of all the collected statistics[2]. RSC enables the hardware to automatically merge packet payloads for common flows and therefore reduce header processing, therefore, this can reduce overall bytes transmitted and received along with potential acknowledgement packets in the process. Further, the lowered instruction count indicates the effect of reduced packet header processing due to RSC packet merging.

[2]We examined other QPS loads and saw similar behavior.

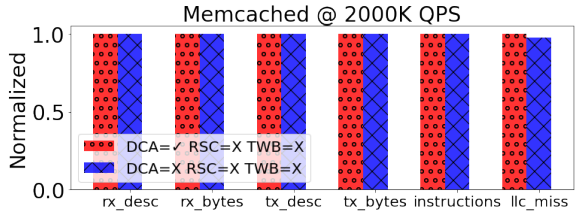

Figure 7: Collected statistics for TSO single and normalized against DCA enable.

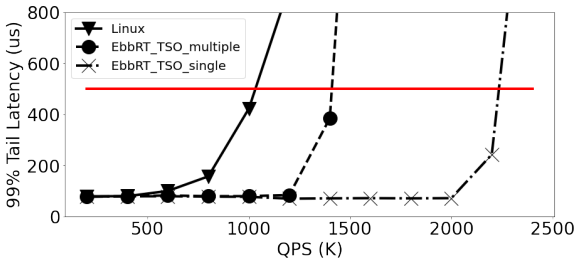

Figure 8: This figure shows Linux and two baseline EbbRT results: DCA, RSC, and TWB are all **disabled**. The red line denotes 500 us tail latency threshold.

4.4 Direct Cache Access (DCA)

Enabling DCA improved performance with TSO single by 4.5% and lowering performance in TSO multiple by 16%. In contrast to fig. 6, fig. 7 shows the differences when DCA is enabled for 2000K QPS. Fig. 7 shows the effect of enabling DCA where it suffered a 2% higher LLC cache miss rate, likely to due cache injections by the NIC. We examined other collected statistics at different QPS loads for both TSO single and multiple case and found that there was a 2%-8% increase in LLC cache miss rate across the board when DCA was enabled. However, it is difficult to pinpoint the exact reasons why the increased LLC miss rate resulted in different performance behaviors - perhaps being able to collect the types of data being evicted may shed more light.

5 Enabling NIC features in Linux

Using EbbRT as a platform for a more guided optimization of the NIC features, we enabled some form of TSO single, DCA, and RSC in Linux:

1. TSO single: Linux's 82599 driver will normally allocate a single transmit descriptor per page size of payload along with another descriptor for the socket buffer header; we were able to force Linux to always use a single descriptor by disabling scatter/gather IO as a 82599 device feature.

2. DCA can be toggled on/off via a Linux configuration build option for the 82599 driver.

3. RSC can be enabled through *ethtool* program.

We found enabling and disabling these features did not impact the peak memcached performance of Linux as it remained relatively stable.

These results indicate that the software overheads of a general purpose Linux makes the potential benefits of these NIC features less discernible compared to a small library OS customized for a single application. For example, the entire EbbRT system running memcached with the 82599 device driver consists of only around 20K lines of code. Further, table 1 also show that there is a limit to performance improvement by enabling these features in combination with each other. As these features are only applicable in the processing of packets in the NIC ring, they only form a part of the overall workload that include other work such as network stack and application processing along with other subsystems such as memory allocation, etc.

Fig. 8 illustrates the performance of Linux and EbbRT with TSO single and multiple configurations. Linux's memcached server runs off a Debian 10.4 distribution with a 5.5.17 kernel. To reduce system noise, we pin memcached to physical cores, disable hyper-threads, TurboBoost, and also disable Linux's `irqbalance` subsystem and affinitize packet receive interrupts to their respective cores. EbbRT's original memcached results when run inside a qemu-kvm (2.5.0) VM [24] was able to achieve 1.3X higher throughput than Linux on 6 cores and the baremetal version in this paper is an extension of that result by using the same memcached source code and runs with 16 cores and obtained 1.4X and 2.2X higher throughput for TSO multiple and single respectively.

6 Conclusion

This paper conducts an experimental study in enabling four hardware features in 82599 NIC that can speed up packet processing. These experiments are conducted for memcached workload using the EbbRT library OS. Selectively enabling these features in EbbRT resulted in a 1.71X performance increase with no modifications to software. Hardware and software statistics are collected to better explain how these features impact performance.

References

[1] Amazon ElastiCache. https://aws.amazon.com/elasticache/.

[2] Atikoglu, Berk and Xu, Yuehai and Frachtenberg, Eitan and Jiang, Song and Paleczny, Mike. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling*

of Computer Systems, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 49–65, USA, 2014. USENIX Association.

[4] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. Dynsleep: Fine-grained power management for a latency-critical data center application. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, page 212–217, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689. USENIX Association, July 2020.

[7] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, aug 2004.

[8] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[9] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 135–148, USA, 2012. USENIX Association.

[10] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, page 50–59, USA, 2005. IEEE Computer Society.

[11] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html.

[12] J. Leverich. Mutilate: high performance memcached load generator. https://github.com/leverich/mutilate.

[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Jonathan Corbet. Large receive offload. https://lwn.net/Articles/243949/.

[15] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 87–98, New York, NY, USA, 2012. ACM.

[16] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[17] Netflix Technology Blog. LevelDB. https://netflixtechblog.com/evolution-of-application-data-caching-from-ram-to-ssd

[18] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 361–377, USA, 2019. USENIX Association.

[19] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Rajesh Nishtala and Hans Fugal and Steven Grimm and Marc Kwiatkowski and Herman Lee and Harry C. Li and Ryan McElroy and Mike Paleczny and Daniel Peek and Paul Saab and David Stafford and Tony Tung and

Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[21] Matthew J. Renzelmann and Michael M. Swift. Decaf: Moving device drivers to a modern language. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 14–14, Berkeley, CA, USA, 2009. USENIX Association.

[22] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 661–676, Berkeley, CA, USA, 2014. USENIX Association.

[23] Sanjay Ghemawat, Jeff Dean. Evolution of Application Data Caching : From RAM to SSD. https://github.com/google/leveldb.

[24] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. Ebbrt: A framework for building per-application library operating systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 671–688, GA, 2016. USENIX Association.

[25] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 119–132, New York, NY, USA, 2011. ACM.

[26] Twemcache: Twitter Memcached. https://github.com/twitter/twemcache.

[27] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021.

[28] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 43–56, Denver, CO, June 2016. USENIX Association.