Flying Memcache: Lessons Learned from Different Acceleration Strategies

Dimitris Deyannis
FORTH
Greece, GR-700 13
deyannis@ics.forth.gr

Lazaros Koromilas
FORTH
Greece, GR-700 13
koromil@ics.forth.gr

Giorgos Vasiliadis
FORTH
Greece, GR-700 13
gvasil@ics.forth.gr

Elias Athanasopoulos
FORTH
Greece, GR-700 13
elathan@ics.forth.gr

Sotiris Ioannidis
FORTH
Greece, GR-700 13
sotiris@ics.forth.gr

***Abstract*—Distributed key-value and always-in-memory store is employed by large and demanding services, such as Facebook and Amazon. It is apparent that generic implementations of such caches can not meet the needs of every application, therefore further research for optimizing or speeding up cache operations is required. In this paper, we present an incremental optimization strategy for accelerating the most popular key-value store, namely `memcached`.**

First we accelerate the computational unit by utilizing commodity GPUs, which offer a significant performance increase on the CPU-bound part of `memcached`, but only moderate performance increase under intensive I/O. We then proceed to improve I/O performance by replacing TCP with a fast UDP implementation in user-space. Putting it all together, GPUs for computational operations instead of CPUs, and UDP for communication instead of TCP, we are able to experimentally achieve *20 Gbps line-rate*, which significantly outperforms the original implementation of `memcached`.

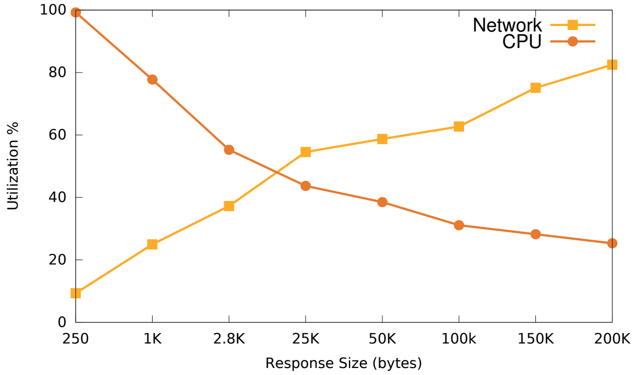


Fig. 1. Network I/O and CPU utilization of `memcached`. The objects size affects significantly the overall performance. Specifically, the performance bottleneck shifts from the network I/O to the CPU, for objects that are greater than 25 KB.

I. Introduction

Modern Internet applications have to serve millions of clients simultaneously, while it has been shown that most of the clients behave as consumers most of the time [1]. Obviously, the addition of new data in Internet applications is disproportionally smaller compared to fetching. As a result, the idea of keeping frequently-accessed data objects in a distributed memory-based cache in order to speed up querying and data fetching appears very attractive.

One of the most popular implementations of such a distributed memory-based cache is `memcached` [2]. [1] Originally implemented for the needs of LiveJournal, the software is now employed by the most popular applications, such as YouTube, Wikipedia, Twitter, and Facebook. `memcached` is based on simple, yet powerful, foundations. The cache is essentially composed by a large key-value store, resident always in memory, and it basically supports two simple operations, `get` and `set`. [2] The communication model is a typical client-server one. Once a client needs to fetch data, it communicates a key to a `memcache` server, the server quickly locates the object in memory, and if it exists it sends the object back to the client, otherwise the fetch involves locating the object on the disk and transferring it to memory. The cache never accesses directly the disk. Once an object is modified in the disk, then the key-value store has to be updated by the application.

The software in its original form can substantially optimize data accessing. However, not all applications have the same requirements and, more importantly, data can substantially differ from service to service. Therefore, the need to optimize the key-value store itself is emerging. Currently, `memcached` has been designed for taking advantage the most widely adopted technologies available. All computation is based on commodity CPUs and networking operations are transferred using TCP. Although these choices seem reasonable enough, in certain cases they can cause bottlenecks, either in CPU or in network I/O. For instance, as we can see in Figure 1, the performance bottleneck is shifted from network I/O to CPU, when the responses size increases.

In this paper, we explore the space of optimizations for `memcached`. Initially, we start by utilizing GPUs to speed up computation. Our approach offers significant performance gains for the computational part of `memcached` but unfortunately under intensive network I/O the modified GPU-based `memcached` is again constrained. Therefore, we proceed to replace TCP with UDP, which significantly boosts performance. Not satisfied by the achieved acceleration we push a step further and use `netmap` [3], a user-space UDP imple-

[1] We usually refer to `memcached` as the software program, which provides the cache, namely `memcache`.

[2] A richer operation set is supported, which we omit here for brevity.

TABLE I. COMPARISON OF THE MODIFIED PROTOTYPE PRESENTED IN THIS PAPER WITH SIMILAR RESEARCH EFFORTS FOR ACCELERATING memcached.

Platform	KOps/s	Lat. (msec)	Watt	Design
TilePRO [5]	1340	0.2–0.4	231	FPGA
Intel Xeon [6]	410	30	143	6-core CPU in-kernel TCP/IP stack
Intel Atom [6]	58	800	35	2x SMT CPU in-kernel TCP/IP stack
MegaPipe [4]	1000	2.2	-	2x 4-core CPUs new networking I/O
TSSP [6]	282	-	16	SoC
This work	1665	0.6	188	CPUs/GPUs user-level UDP stack

mentation which avoids redundant copies to kernel space. This results in dramatic improvements of memcached in terms of throughput and latency. Our prototype is able to outperform the original implementation of memcached by achieving a 20 Gbps line-rate. Similar speedups for memcached have been reported with MegaPipe [4], and therefore our paper *experimentally validates* a maxima in memcached acceleration, which is mainly driven by boosting network I/O, and independently confirms a past architectural design.

Our optimization strategies revealed a lot of interesting insights, which we summarize in the following contributions:

- The benefits stemming from using GPUs in memcached are not significant. A commodity GPU outperforms a high-end CPU in parallel computation, such as hashing, something that we demonstrate in this paper using three different GPU models. However, the computational overhead in memcached is low. Therefore, a design choice of replacing CPUs with GPUs for the computational part is not justified in the context of memcached. Nevertheless, the energy and cost savings related to using GPUs instead of CPUs for memcahed can make the GPU choice more attractive.
- As previously reported, we verify that memcached dominant overhead is due to network I/O. By replacing the TCP stack with a custom *user-space* UDP stack implementation we are able to achieve *20 Gbps line-rate*, and therefore, we *experimentally validate* a maxima in memcached acceleration – mainly driven by boosting network I/O – as it was shown in [4].

II. PREVIOUS WORK

Many services leverage memcached as a caching layer between a web server tier and back-end databases. Such applications are demanding, therefore a series of changes are needed for fully taking advantage of the caching system. Table I provides a summary of the various research efforts into accelerating memcached.

The most common approach is optimizing memcached using reconfigurable hardware [5]–[8]. These systems offer a scalable method of processing incoming requests in high-speed environments. However, most implementations require specialized programming, and are usually tied to the underlying device. Lim et al. [6] differentiate their work by relying on hardware to only accelerate the common case (i.e., serving of the GET requests), while relying on software for the remaining memcached features.

Other studies have focused on improving the network I/O of memcached using either non-commodity networking hardware like Infiniband and RDMA interfaces [9], or software-based RDMA over commodity network cards [10]. Compared to these approaches, which can push the performance load to the clients, our design supports the standard memcached interface with no modifications. MegaPipe [4] provides a new programming interface for high-performance networking I/O, targeting applications that typical use persistent connections to servers, such as memcached.

Recently, graphics cards have provided a substantial performance boost to many network-related workloads [11]–[16]. The first work that utilized GPUs to offload the hashing computations of memcached is [17]. Both discrete and integrated GPUs were used, however their evaluation did not cover the networking I/O operations, which is a significant bottleneck for memcached performance.

III. ACCELERATION STRATEGIES

Our architecture comprises of commodity hardware only, including one or more CPUs, a discrete GPU and one or more 10 GbE NICs, which communicate for handling incoming requests. The CPU analyses incoming packets to extract key, value and metadata information. Both ASCII and binary protocols are supported, while additional protocols can be easily added without any performance penalty. The information is passed to the GPU which produces an index into the value store for any given key. The value store supports read or write operations: in SET operations, the corresponding value is written into the store, while for GET operations, the retrieved value is added to the response (comprised of one or more packets) that is forwarded back to the client.

The practical challenges of mapping the aforementioned operations to commodity hardware are (a) support for high performance hashing computations, and (b) minimizing the execution paths in the networking stack. These observations lead us to two design principles: (i) key hashing is offloaded to massively parallel many-core computational devices (GPUs), and (ii) networking stack is moved to user-space, to reduce data movements and context switches.

A. *Offloading Hashing to the GPU*

Figure 2 depicts the workflow of the design for the GPU-assisted memcached. In this initial design, we modified the original memcached to offload all hashing computations to the GPU. memcached uses libevent [18] for asynchronous I/O, and maintains different threads (fixed to four by default) for processing incoming connections. Specifically, a single thread is responsible for receiving incoming connections via the traditional socket() interface and distribute them to the remainder active threads on a simple round-robin basis; upon assigning to a thread, the connection is served solely by that thread until its completion.

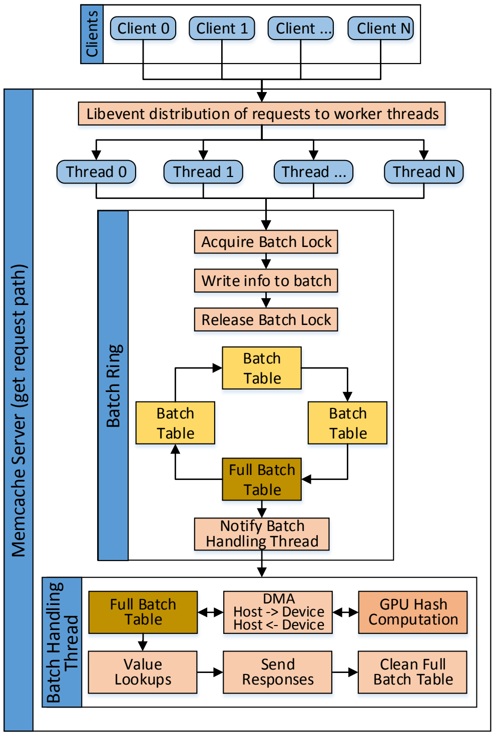


Fig. 2. Workflow architecture of the GPU-assisted `memcached` for handling GET requests.

We modified the default execution flow for leveraging the massively parallel compute capabilities of GPUs. Specifically, each request is analyzed by its assigned thread and the protocol's command is extracted. In the case of a GET command, the corresponding key is stored in a shared buffer which is processed in parallel by the GPU every time it fills up. For this, a separate thread is spawned, which is responsible for passing the filled buffer to the GPU, launching the GPU kernel that hashes the keys, and transferring the resulting hash checksums from the GPU memory back to the host memory. Thereafter, the host thread iterates through computed hashes and performs the necessary lookups to the hash table, to find the requested response objects, if any. The aforementioned objects are crafted in appropriate responses that are sent back to the client via the network socket that received the corresponding request. To further improve parallelism, we used a double buffering scheme. When the first buffer becomes full, it is transferred to the global memory of the GPU that can be read later through the kernel invocation. While the GPU is performing computations on the keys of the first buffer, the CPU will copy newly arrived keys in the second buffer.

To compute the key hashes of a single buffer in the GPU, a number of threads equal to the size of the buffer is created. As the hashing algorithm used by `memcached` cannot be parallelized internally – for example by splitting a single key to different portions and assigning each portion to a different thread – a single key is assigned to a separate thread. As we will see in Section IV-C, the larger the buffer, the more the parallelism that will be exposed by the GPU. Each thread executes the hashing algorithm – we have ported the same hash function used by the original version of `memcached` – to its input key. An important optimization of the GPU execution is related to the way the input keys are loaded from the device memory. Since the input symbols belong to the ASCII alphabet, they are represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput would be reduced by a factor of up to 32. To utilize the memory more efficiently, we redesigned the input reading process such that each thread is fetching 16 bytes at a time, using the `int4` built-in data type, instead of one byte. When a thread computes the hash of its assigned key, it copies the resulted 32-bit checksum in an array allocated in device memory. After the computation is completed, the array with the hash checksums is transferred back to the host's memory.

Each 32-bit checksum is used as an index to a lockless hashtable – we use the hashtable implementation used by the original `memcached`. The objects that are stored in are multi-versioned internally and reference counted, hence no client can block any other client's actions. In addition, the hashtable is implemented as an array of buckets. Each bucket contains a list of nodes, with each node containing a $(key, value)$ pair. Upon a match, the full key, which is stored in the hash table entry, is compared to the requested key, and if they match the corresponding metadata is updated, for example the last-access timestamp used for LRU replacement. If no match is found, a miss response is sent.

B. *Speeding Up Network I/O*

A significant portion of the overall execution is spent on network I/O. To optimize the poor performance of the Linux networking stack we leveraged several common characteristics of the `memcached` behavior. Particularly, we focused on accelerating GET requests, since they vastly outnumber SET requests in real-world scenarios [5], [19]. To that end, we used UDP, instead of TCP, for handling GET requests, in-line with the majority of previous research [6], [7].

UDP is a much lighter protocol than TCP, because it offers more relaxed packet delivery guarantees. Typically it is used for operations that are not required to succeed, such as GET requests where a missing or incomplete response can simply be treated as a cache miss; requests that must be reliable, like SET operations, should be transmitted using TCP.

Switching from TCP to UDP, offers a clear throughput advantage, that can be explained by the fact that TCP is a transaction-based protocol that suffers additional overhead, especially for small packets [5], [6]. Still, a lot of CPU cycles are spent on the OS networking stack, as we verify in Section IV. The main reason behind this is the complicated memory management and the several copies that occur for each data sent or received. Traditionally, networking applications use the socket interface to allow messages to be transmitted between hosts. Each message is passed through a multi-layer networking stack (UDP, IP, and Ethernet protocols), and, at any level of the stack, the associated protocol encapsulates the message into its own format, adding a header containing extra control information. Although the networking stack abstraction simplifies development, the costs in terms of performance when operating in multi-10Gbps environments are not negligible. The main reason behind that is the frequent per-packet

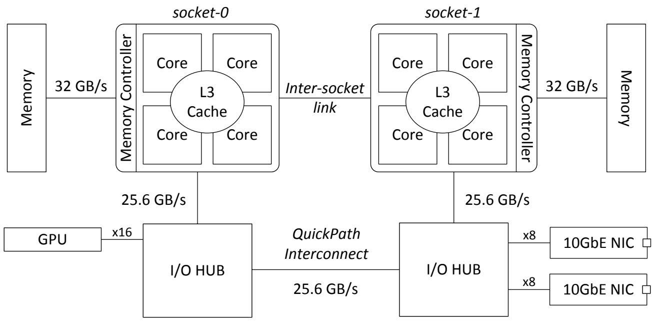


Fig. 3. The high-end setup used for carrying out the evaluation of the custom `memcache` prototype.

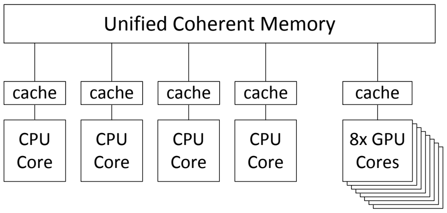


Fig. 4. The low-end setup used for carrying out the evaluation of the custom `memcache` prototype.

buffer allocations and deallocations that stress the memory subsystem of the kernel, as well as the context switches that occur every time a new packet is received [20].

1) User-space UDP Stack: As described above, excess data movements and context switches cause significant CPU overhead. To avoid the problem, we leveraged a user-space UDP stack implementation, which completely bypasses the kernel. In our design, all of the packet processing and framing occurs inside the context of the `memcached` application. Specifically, we have modified `memcached` for interacting with the network interface directly, and send or receive fully-formed packets (including the UDP/IP headers) directly to the network interface. To achieve this, we utilized the `netmap` module [3], which gives userspace applications a very fast channel for exchanging raw packets with the network adapter. The `netmap` module maps the Rx and Tx packet buffers, which are allocated initially at start-up for performance reasons, and all necessary metadata structures to the process address space. As a result, a user-space application can easily receive incoming packets by polling on the mapped Rx packet buffers. In case a new packet is received it is passed to the upper layers for analysis.

In the current `memcached` implementation, requests must be contained in a single UDP datagram [21]. The only common requests that would span multiple datagrams are huge multi-key GET requests and SET requests, both of which are more suitable to TCP transport for reliability reasons [21]. As such, we do not need to keep any state in the receive side to reassemble requests that span multiple packets. On the transmission side, the program simply writes the corresponding raw packets (with the appropriate UDP and IP headers) to the mapped Tx buffers. In case the response spans several datagrams, we contain a simple frame header in each UDP datagram, right before the actual data. The frame header, defined in `memcached` protocol [21], is 8-byte long and contains the request ID (as supplied by the client), the sequence number of the current datagram, and the total number of datagrams for this message. Using the data provided in the frame header, the client is able to keep track and reassemble the full response correctly.

IV. Performance Evaluation

We evaluate the performance of our system, using a variety of workloads and hardware setups. We first describe the experimental testbed (Section IV-A) and the corresponding workloads (Section IV-B). We then analyze the sustained performance under different scenarios using micro-benchmarks (Section IV-C), as well as high-level end-to-end performance measurements (Section IV-D). We also provide an energy and cost analysis (Section IV-E).

A. Systems under test

We perform all the experiments using two radically different server setups: a high-end Xeon-based server and a low-end, low-power AMD APU-based server. Both setups are summarized in Table II.

The high-end system is equipped with two Intel Xeon E5520 2.27 GHz NUMA nodes, each of which contains four cores. This totals to 8 physical threads or 16 virtual threads with SMT enabled. Each NUMA node is equipped with 6 GB of memory, and is also connected to the peripherals via a separate I/O hub, linked to multiple PCIe slots (see Figure 3). For our network needs, we utilize two Intel 82599EB 10-GbE network adapters, both placed on the same domain. We also place a GTX 770 GPU on the second domain; after experimentation, we found that it is better to keep the NIC packet buffers and the GPU to remove domains.

The APU-based system represents a low-power alternative to the Xeon-based setup and its main purpose is to improve the energy efficiency within a fixed data center power budget. It is equipped with a AMD A10-7850K APU with Radeon R7 Graphics running (shown in Figure 4) at 3.7 GHz and 6 GB of memory. We also utilize two Intel 82599EB 10-GbE NICs.

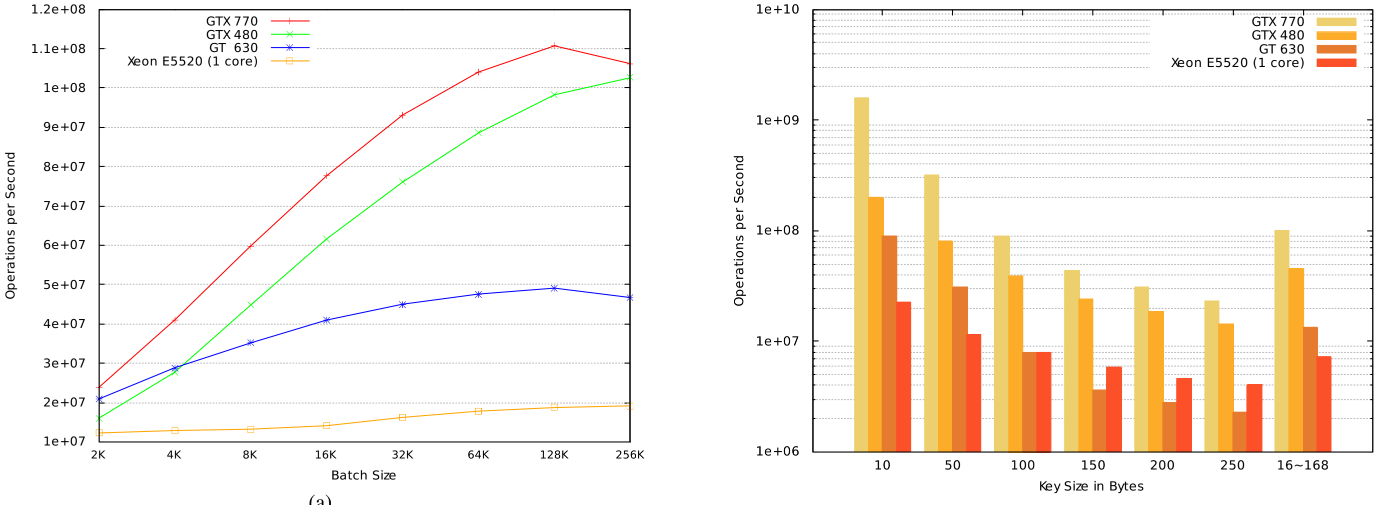
We note that even though the two systems are equipped with different amounts of memory (12 GB in the high-end against 6 GB in the low-end), memory capacity has no direct effect neither on latency nor on throughput because the share of a cluster's overall load directed to a particular node is proportional to the node's share of the overall cluster memory capacity. Both systems ran Linux 3.5, with CUDA v5.0 and Intel SDK for OpenCL Applications XE 2013 installed. We also used our custom version of `netmap` [3].

B. Workloads

The behavior of `memcache` varies considerably based on the size of the keys and values it stores; the actual contents of both keys and values are opaque byte strings and therefore do not affect the behavior of the system. To that end, we design four different synthetic workloads that represent a wide range of possible key-value samples. Our aim is to create a set of realistic workloads that expose the sensitivity of `memcached` performance in the context of particular applications.

TABLE II. SYSTEMS UNDER TEST. WE USED TWO DIFFERENT SYSTEM ARCHITECTURES. A XEON BASED CPU SERVER WITH DISCRETE GPU AND A LOW-POWER APU BASED SERVER WITH INTERGRATED GPU.

Components	**High End**	**Low End**
Processor	2x Intel Xeon E5520 @ 2.27GHz	1x AMD A10-7850K APU with Radeon R7 Graphics @ 3.7 GHz
DRAM	2x banks 6 GB DDR3 (NUMA) @ 1066 MHz	1x bank 6 GB DDR3 @ 1333 MHz
10-GbE	2x Intel 82599EB 10-GbE NICs	2x Intel 82599EB 10-GbE NICs



(a)

(b)

Fig. 5. Performance sustained for hash operations for different computational devices, as a function of (a) keys processed at once (batch size) and (b) key size.

TABLE III. WORKLOAD CHARACTERISTICS. WE CONSTRUCTED DIFFERENT WORKLOADS TO EXPOSE THE BROAD RANGE OF `MEMCACHED` BEHAVIOR.

Name	**Avg. Size**	**Description**
FixedSize	250 B	Fixed objects that place the greatest stress on `memcached` performance [6].
MicroBlog	1 KB	Queries for short snippets of text, like for example user updates, similar to the size of *tweets* used by Twitter.
Wiki	2.8 KB	Represents portions of articles found in Wikipedia.
ImgPreview	25 KB	Comprises of photo objects (thumbnails) found in photo-sharing sites.

1) Key Selection: For typical use-cases the key-size was between 6 to 168 bytes with an average of 30.7 bytes as was recently reported for several production clusters at Facebook [19]. We use the same key-size distribution for all of our experiments.

2) Value Selection: We design four different value-workloads that represents a wide range of possible scenarios adapted from [6]. Each one is defined by an object-size distribution which represents realistic workloads for a particular application assisted by a `memcache`. We notice that even though we synthetically create the workloads, they maintain the same notation and attributes introduced by earlier work [6]. Table III briefly re-introduces the reader to the workload characteristics.

a) FixedSize: The simplest workload uses a fixed object size of 250 bytes and uniform popularity distribution. We include this workload because small objects place the greatest stress on `memcached` performance [6].

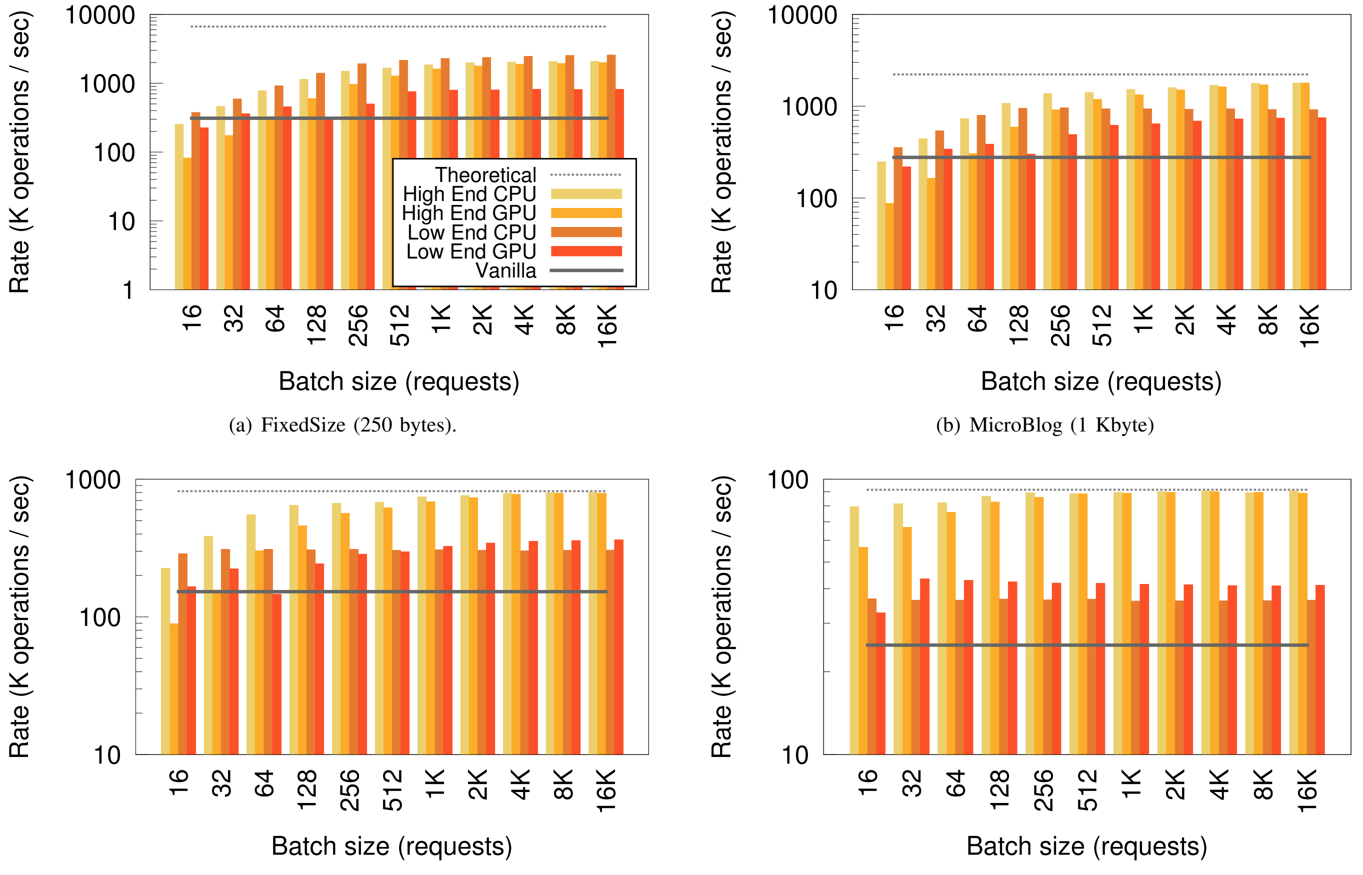
b) MicroBlog: This workload comprises queries for short snippets of text, like for example user updates, similar to the size of *tweets* used by Twitter. The text of a tweet is restricted to 140 characters, however the average size grows to approximately 1 KB when associated meta-data are included.

c) Wiki: This workload represents portions of articles found in Wikipedia. Particularly, each object represents an individual article in HTML format of an average size equal to 2.8 KB.

d) ImgPreview: This workload comprises of photo objects found in photo-sharing sites. Typically, photo sharing sites use `memcached` to serve thumbnails; larger files of high-resolution images are served from other data stores (see for example Facebook's Haystack system [22]). As such, the average object size for this workload is 25 KB.

C. Micro-benchmarks

We first measure the key hashing throughput when executing on the GPU. We use three different GPU models, each of which had different energy and computational characteristics. Specifically, we utilize a low-end NVidia GT 630, a high-end NVidia GTX 480, and a high-end NVidia GTX 770. Figure 5(a) shows the overall hashing throughput achieved for each device. We observe that the performance of GPUs increases proportionally to the number of keys that are processed at once (batch size), reaching a plateau at 128K keys. For comparison, we also plot the performance achieved by a single CPU core of the high-end setup, shown in Table 3. All



(a) FixedSize (250 bytes).

(b) MicroBlog (1 Kbyte)

(c) Wiki (2.8 Kbytes)

(d) ImgPreview (25 Kbytes).

Fig. 6. Overall performance sustained for the workloads presented in Table III. The high-end setup is able to reach theoretical rate for Wiki and ImgPreview workloads with batch size larger than 512.

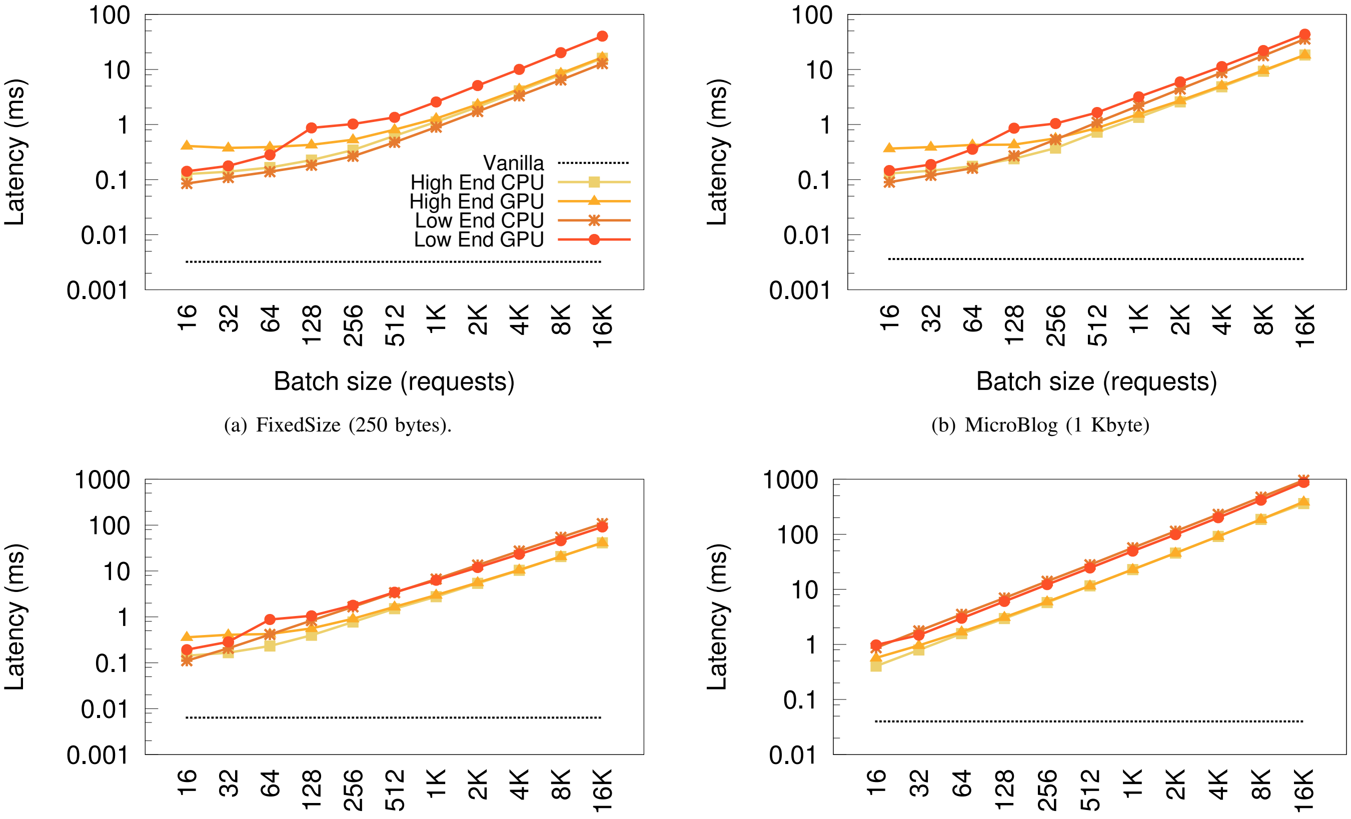
GPUs perform better compared to the CPU, however they have different behavior. For instance, high-end GPUs are able to sustain up to 12.8 times better compared to the CPU, with GTX 770 being about 10% better than GTX 480 in the common case scenario. The performance of the low-end GT 630 lies in the middle. We notice that GT 630 has worse performance than the CPU for keys larger than 100 bytes. Nevertheless, 200- and 250-byte long keys is the worst case scenario, and GT 630 outperforms the CPU in the typical key sample reported in [19].

We also explore the performance of hash computations using a set of seven different key workloads. The first six workloads contain fixed sized random alphanumeric keys with sizes of 10 up to 250 bytes, which is the maximum key size according to memcached semantics. In order to increase parallelism on the GPU, we process 128K keys at once. Figure 5(b) shows the performance for each device. As the key size increases the throughput of the system drops, as expected, due to the heavier computations needed. We also use keys of various lengths produced randomly from a random distribution (last bar), in order to measure how the irregularities in the key lengths—that occur in a real case scenarios— affect the GPU performance. Fortunately, GPU execution is able to hide such discrepancies, without requiring extra scheduling or more sophisticated placement.

D. Macro-benchmarks

In this section we evaluate how our proposed modifications perform in a real-network setup, using the four different workloads that are described in Section IV-B. Keys are random strings of size between 6 to 168 bytes with an average of 30.7 bytes, again, as was recently reported for several production clusters at Facebook [19]. A series of GET requests are generated from two clients with 10-GbE network interfaces, connected back-to-back with the base machine. Each time, memcached is initialized with 10,000 objects, which the clients request in a round-robin fashion; this way we stress the hashtable lookup mechanisms, while the CPU caches were outgrown, forcing access to memory on the critical path. This is important because memory references create traffic on the interconnecting links and may sometimes interfere with the overall system's performance in unexpected ways. For comparison, we also plot the performance of the vanilla implementation.

Figures 6(a)- 6(d) show the overall performance (measured as operations that completed successfully per second) for each of the workloads described previously. We also show the corresponding latency in Figures 7(a)- 7(d). In our implementation, we utilize four CPU cores (out of the eight): two CPU-cores were used to handle the reception and transmission of data



(a) FixedSize (250 bytes).

(b) MicroBlog (1 Kbyte)

(c) Wiki (2.8 Kbytes)

(d) ImgPreview (25 Kbytes).

Fig. 7. Overall latency sustained for the workloads presented in Table III. Latency form stabilazes for workloads higher than 2.8 Kbytes.

(one per network port), and two CPU-cores were used for serving (i.e. hashing and lookup) each connection. We used the same number of cores in vanilla memcached for a fair comparison. Please note that by increasing the size of objects, the performance bottleneck shifts from the computing units to the network. Particularly, as the size of objects increases, the sustained throughput approaches the maximum line-rate capacity. In contrast, when the size of objects decreases we see that the network performance becomes less and less important because the corresponding compute device becomes dominant. By increasing the number of keys (batch size) that are hashed at once, both CPU and GPU execution benefits in terms of throughput. Due to the small size of the keys, assigning more computations to each computing unit, pays off the static cost of threads creation, resulting to about 1.47–5.71 times speedup compared to the vanilla memcached. We note that the batching of keys results to increased latency, that grows proportionally to the size of batch. Still, using a batch size of around 512 requests effectively balances the trade-off between throughput and latency in most cases. Leveraging a GPU to offload hashing does not pay off, due to the extra data transfers involved, over the PCIe bus.

E. Energy benefit

We evaluate the power consumption characteristics and the price of the components of our system. The power reported here is measured using an external meter, without applying any frequency/voltage scaling. In the high-end setup the Xeon core consumes 10–15 W as indicated in Table IV. The GT 630 GPU consumes a total of 50 W when active, that is comparable to four Xeon cores that consume about 48 W. Higher performance alternatives are the GTX 480 and GTX 770 GPUs with 123.5 W and 134.5 W respectively. The low-end setup utilizes the intergrated GPU of the A10-7850K APU thus not needing a discrete GPU. The total consumption of the APU is 95 W on full load which is 242% lower than the high-end setup in terms of performance and 529% lower in terms of purchase cost.

V. Limitations

Since netmap maps the network interface to the process, bypassing the kernel, it is implied that the NIC must be used solely by memcached. We anticipate that the design proposed in this paper will be used by demanding applications which consider the operation of memcache critical, and therefore, dedicated hardware can be afforded for this operation. In more complicated setups where memcache serves more than one key-value store, certain network packets might need to be handled by other processes. Relaying particular network packets to the kernel for further delivery is possible and it is addressed in the original implementation of netmap [3].

TABLE IV. PER-DEVICE POWER CONSUMPTION AND COST.

Device	Power	Cost
Intel Xeon E5520		250 $
first core	+15 W	
subsequent core	+11 W	
AMD A10-7850K APU	95 W	170 $
NVIDIA GTX 770		400 $
idle	23.5 W	
active	134.5 W	
NVIDIA GTX 480		500 $
idle	48.5 W	
active	123.5 W	
NVIDIA GT 630		60 $
idle	8 W	
active	23.5 W	
Intel 82599EB 10-GbE	7 W	430 $

Moreover, the modified prototype presented in this paper is based on UDP. Therefore, to support operations which require reliability such as Set or Update, we have to either change the `memcached` protocol and add logic to it, or implement TCP in user-space. Fortunately, there are recent promising results towards implementations of TCP in user-space [23].

VI. Conclusion

We have applied different acceleration strategies for optimizing the performance of `memcached` in terms of throughput, latency and power consumption. We initially started by utilizing GPUs to speed up computation, which gave us significant performance gains, but unfortunately – in case of intensive network I/O – `memcached` was again critically constrained. We proceeded and replaced TCP with UDP, which boosted performance significantly. Not satisfied by the achieved acceleration we pushed a step further and used `netmap` [3]. Our prototype was able to outperform vanilla `memcached` by recording 20 Gbps line-rate.

VII. Acknowledgments

This work was supported by the General Secretariat for Research and Technology in Greece with the Research Excellence grant, GANDALF.

References

[1] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI, 2013.

[2] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 2004, no. 124, Aug. 2004.

[3] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC, 2012.

[4] 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, 2012.

[5] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Proceedings of the 2011 International Green Computing Conference and Workshops*, ser. IGCC, 2011.

[6] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA, 2013.

[7] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10Gbps Line-rate Key-value Stores with FPGAs," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, ser. HotCloud, 2013.

[8] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached Appliance," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA, 2013.

[9] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. Islam, X. Ouyang, H. Wang, S. Sur, and D. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," in *Parallel Processing (ICPP), 2011 International Conference on*, ser. ICPP, 2011.

[10] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached," in *Proceedings of the 2012 USENIX Annual Technical Conference*, ser. USENIX ATC, 2012.

[11] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID, 2008.

[12] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID, 2009.

[13] W. Sun and R. Ricci, "Fast and Flexible: Parallel Packet Processing with GPUs and Click," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '13, 2013.

[14] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: a multi-parallel intrusion detection architecture," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, ser. CCS, 2011.

[15] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-Accelerated Stateful Packet Processing Framework," in *Proceedings of the 2014 USENIX Annual Technical Conference*, ser. USENIX ATC, 2014.

[16] L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis, "Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures," in *Proceedings of the 10th ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS, 2014.

[17] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ser. ISPASS, 2012.

[18] "libevent - an event notification library," http://libevent.org.

[19] 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, 2012.

[20] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM, August 2010.

[21] "memcached protocol," https://github.com/memcached/memcached/blob/master/doc/protocol.txt.

[22] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2010.

[23] 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, 2014.