Implementing Ultra Low Latency Data Center Services with Programmable Logic

John W. Lockwood, Madhu Monga
Algo-Logic Systems, Inc., Santa Clara, CA 95050
JWLockwd@Algo-Logic.com, Madhu@Algo-Logic.com

***Abstract*—Data centers require many low-level network services to implement high-level applications. Key-Value Store (KVS) is a critical service that associates values with keys and allows machines to share these associations over a network. Most existing KVS systems run in software and scale out by running parallel processes on multiple microprocessor cores to increase throughput.**

In this paper, we take an alternate approach by implementing an ultra-low-latency KVS in Field Programmable Gate Array (FPGA) logic. As with a software-based KVS, lookup transactions are sent over Ethernet to the machine that stores the value associated with that key. We find that the implementation in logic, however, scales up to provide much higher search throughput with much lower latency and power consumption than other implementations in software.

As with other KVS systems like redis, memcached, and Aerospike, high-level applications store, replace, delete, and search keys using a standard Application Programming Interface (API). Our API hashes long keys into statistically unique identifiers and maps variable-length messages into a finite set of fixed-size values. These keys and values are then formatted into a compact, binary Open Compute Storage Message (OCSM) format and transported in User Data Protocol (UDP)/Internet Protocol (IP) Ethernet frames over 10 Gigabit/second Ethernet (10GE) or faster network links.

When transporting OCSM over 10 GE and by processing the performing the key/value search in FPGA logic, a fiber-to-fiber lookup latency of under a half microsecond (0.5 µS) was achieved. Using four 10 GE interfaces, a single instance of the FPGA-accelerated search core achieves a throughput of 150 Million Searches Per Second[1] (MSPS). Compared to traditional software, the FPGA KVS is 88 times faster while using 21x less power than socket I/O. Compared to software optimized with DPDK kernel bypass, the FPGA KVS was measured to process messages 10x faster while using 13x less energy.

Keywords—Key Value Store, KVS, NoSQL, SDN, Ethernet, datacenter, DPDK, GDN, FPGA, latency, power, throughput.

I. Introduction

Association of a key to a value is an essential network service in a modern data center. Keys refer to alphanumeric strings or binary data. Values are associated with keys and they may change over time. Key Value Store systems are widely deployed to share data between application servers as users shop, trade, and share data over the web. Examples of widely used KVS deployments include Dynamo at Amazon [DynamoDB] and Memcached at Zynga, Twitter, and Facebook [Memcache-FB].

KVS systems are also used by telecommunication carriers to track the status of mobile users, devices, and network activities. For these purposes, the throughput of the KVS is a critical metric because the number of users and their activities continues to rapidly grow.

Today, millions of servers are used to associate billions of keys with values every second due to human-to-human and human-to-machine interactions. In the future, billions of servers will be needed to handle trillions of machine-to-machine interactions every second. Without a highly efficient KVS implementation, the capital expense to purchase such large numbers of servers and the operating expense to power them with electricity will become a bottleneck that limits the scalability and performance of new services.

In algorithmic trading systems, for example, KVS servers are used to track orders by associating a unique key with each transaction identifier (ID). The value of the bid or ask price of a stock, for example, is determined by performing a lookup with the order ID as the key. Trading servers are co-located in data centers so that they can monitor live market conditions and automatically place orders with exchange servers with the lowest possible latency. For these systems, the time it takes to complete a transaction (the latency) is a critical metric.

A. *Database Services*

Traditionally, the Structured Query Language (SQL) was used as a standard interface to relational databases. SQL queries were transferred over TCP/IP sockets between client machines and database servers to store structured data in the form of fixed rows and columns. Massive growth in the number of mobile users and devices connected to the internet, the increased amount of information captured, and swelling volumes of metadata, has led to massive growth of datasets [McKinsey]. NoSQL databases provide a more efficient means to store, manage, and analyze big data. Whereas relational databases scale by adding memory and using increasingly powerful CPUs, NoSQL databases [Pokorny IJWIS] scale by adding additional servers to handle massive volumes of unstructured data. NoSQL databases have proven to be highly effective for organizing document data, searching graphs, and supporting wide-column stores. In this paper, we focus on a unique approach to improving the throughput while lowering the latency and power of KVS by processing and storing the key-value pairs in programmable logic.

[1] Searches Per Second is equivalent to Requests Per Second as referred by other authors

The shift from the relational database to NoSQL was driven by the need to process big data. Simple NoSQL interfaces make it easy for developers to store and search unstructured data. The increased workload required for large-scale cloud services drove the demand to both scale-up and scale-out KVS systems. Inefficiencies in microprocessor architectures limit the ability of current systems to scale up [Ferdman SIGARCH]. The use of wide and out-of-order instruction executions have been shown to exhibit low instruction-level and memory-level parallelism.

Most existing NoSQL databases are deployed on standard microprocessor-based servers. Datacenter operators have dutifully added more standard servers with additional memory and Ethernet links to meet the demands of increased network traffic. With these standard servers, as analyzed in [Ferdman SIGARCH], performance improvements were sub-linear with the increased workloads. KVS workloads are not well suited for microprocessor cores because they make poor use of the Silicon die area and consume unnecessary power. On the other hand, servers that use logic have been found to perform NoSQL operations better than microprocessor-based servers. For example, significant improvements to memcached were achieved by offloading KVS lookup to FPGA logic [Blott USENIX].

In this work, we contribute a state-of-art FPGA-based KVS that achieves record-breaking performance through the use of a new, open-standard message format that sends compact, binary-encoded key-value pairs over Ethernet and offloads all message processing and lookup to FPGA logic.

B. *Critical Metrics*

Critical metrics for KVS are latency, throughput, and power. Stock traders, brokers/dealers, and option exchanges trading in co-location facilities need to lookup order IDs with very low latency. Consumer search and social networking websites need high throughput to search for keywords and match user IDs to host billions of users. Telecommunication and data communication providers want both high throughput and low latency to lookup phone numbers, Electronic Serial Numbers (ESNs), and Internet Protocol (IP) addresses as increasing volumes of voices and data stream over wired and wireless networks.

The expense to operate a service scales to the amount of power that is required to perform the service. Existing operators of large data centers, such as Google, Facebook, Amazon, Apple (i-Cloud), Microsoft Cloud Services (Azure), and Rackspace, want to quickly deploy scalable services using the least amount of energy. Modern data centers have grown and now use millions of servers and consume hundreds of Megawatts.

By increasing server throughput and reducing the energy required per operation, data center operators can reduce both capital and operating expenses. The cost to implement a data center service is the sum of the capital expense to purchase servers plus the operating expense to run the servers throughout the life of the machines. The minimum number of servers required to implement a service can be computed as the peak throughput of the service divided by the throughput per server. More servers are required for machines that have lower throughput.

Networking services like databases were traditionally implemented as user-space applications running on top of a host operating system. More recently, there has been a trend to more optimally implement networking functions in software by avoiding the overhead of the Linux kernel, interrupts, and context switches. As such, we benchmark KVS implemented by using both traditional software with sockets as well as with an optimized software approach using Intel's optimized Data Plane Development Kit [DPDK].

C. *Contributions*

In this work, we focus on the results achieved by scaling up the performance of KVS. We offload the KVS service to FPGA logic that runs on an off-the-shelf expansion card installed in standard server. Using multiple cards, the KVS service also scales out to handle larger workloads but with far fewer servers.

We introduce an open standard message format, Open Compute Storage Message (OCSM), to encapsulate the key-value pairs in a UDP payload and send them to KVS on an FPGA over the Ethernet. The messages convey similar data to that used by the memcached format, but the message format is structured to reduce the overhead of data parsing and field extraction in hardware.

We implemented two different versions of the FPGA KVS that differ in how key-value pairs are stored: either in on-chip memory of the FPGA or in the off-chip DRAM near the FPGA. The implementation with on-chip memory achieves a record-breaking 10 Gigabit Ethernet-to-10 Gigabit Ethernet latency of only 467 ns to return a value from a key carried in a UDP/IP packet with a 32 byte message. A single instance of the search core in FPGA logic can support a throughput of 40 MSPS over a 10 GE link or 150 MSPS using four 10 GE links.

The second implementation of KVS uses the off-chip memory to search a 64 byte message. For a low-cost hardware device with 8GB of DDR3, a single instance of this implementation can support a throughput of 20 MSPS on a 10GE link and support a table depth of up-to 48M entries. The KVS scales to support higher throughputs, limited only by the number of available Ethernet ports and memory banks. The depth the table scales with the size of the off-chip memories attached to the FPGA.

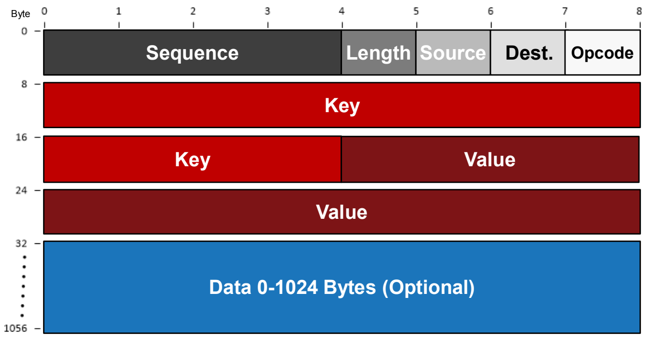
Because OCSM is a new format, other NoSQL databases cannot be directly compared to our implementation. However, throughput results of the popular Aerospike KVS [Aerospike] with a fixed message size indicates that their KVS implementation can perform 7000 queries per second. Each query can fetch 100 messages and each message is 523 bytes long. These operations are equivalent to performing 700K searches per second. To fetch a 523 byte message from our implementation, we perform nine searches with an overall throughput of 2.2 MSPS, which is three times faster than results from Aerospike.

II. OPEN COMPUTE STORAGE MESSAGE (OCSM)

Open standards are critical for the success of data center application deployments. The OCSM is a new, compact, open-standard packet format that is suitable for use in both Intranet and Internet-wide KVS. The length of OCSM for KVS using the FPGA's on-chip memory is 32 bytes and the length of OCSM for KVS using the off-chip memory is 64 bytes. The first six bytes of the UDP payload containing the OCSMs is a header that identifies the number of OCSMs in the packet as well as the type of OCSM being used. For a Maximum Transmission Unit (MTU) of 1,514 bytes, which is common for Ethernet LANs, the UDP payload may contain up to 45 32-byte messages or 22 64-byte messages.

In the current implementation, all messages in a packet must be either an update or search type. The update type includes messages that insert, modify, or delete entries from the KVS. Search is used to perform a lookup in the KVS and return a response.

The request format of an OCSM and the data structure as C code is detailed in Figure 1. Each message holds six 4-byte integer parameters for a 32 byte OCSM or 14 4-byte integer parameters for a 64 byte OCSM, a sequence number, a length, a source, a destination, and an opcode. Larger messages store additional data at the end of the message. Each OCSM has a fixed length of 32 or 64 bytes plus a variable-size data component defined as a multiple of any additional 32 or 64 byte units.



```
// OCSM message structure
typedef struct {
  unsigned int parameter[6]; // Stores Key and/or Value
  unsigned int sequence; // 32-bit sequence #
  unsigned char length; // typically 1
  unsigned char source; // >1 device sources commands
  unsigned char destination; // for multiple OCSM instances
  unsigned char opcode; // Indicate insert / lookup / etc.
  unsigned char data[DATA_LENGTH]; // (len-1)*32 bytes
} csm_t;
```

Figure 1: Format of 32 byte OCSM Request

The OCSM places no limitation on the key and value that can be supported. For an application that requires a key wider than 96 bits, API function calls can hash over the wider keys to generate a 96 bit key. Similarly, for applications that need to store values wider than 96 or 352 bits, multiple locations can be used to store the longer value in the KVS. The latency to look up longer values is not directly proportional to the number of operations. To lookup longer values, the total latency is simply the time required for the first lookup to identify the length of the value plus time required for the second dependent lookup followed by a series of pipelined lookup operations. The throughput and power proportionally scale with the number of operations required per message.

III. IMPLEMENTATIONS OF SCALE UP KVS

KVS can be implemented in software multiple ways. Traditional software implementations are built as a user-space application daemon that communicates with the network operating system via standard BSD-style network sockets. Optimized software implementations avoid the overhead of processing the packets in the kernel by dedicating CPU cores to read packets directly from the Network Interface Card (NIC).

A. Search using Software

Traditional software sockets allow user-space applications to bind and accept network connections as a service provided by the kernel of the operating system. The user-space program implements the key-value network service. This program receives packets with keys from a UDP/IP or TCP/IP network socket and then sends packets back with the value. The kernel, in turn, controls the NIC to send and receive packets on the Ethernet interface.

Using an Intel core i7 4770k 3.4 GHz CPU and an Intel 82598 [Intel82598] 10 GE NIC running on the CentOS (RedHat-based) operating system, we implemented the OCSM-based KVS in C. We used the software socket implementation as a baseline for the software and hardware-accelerated implementations described in the next sections.

B. Search using DPDK

Intel® DPDK is a set of data plane libraries and network interface controller (NIC) drivers for fast data processing on an Intel® Architecture (IA) platform. To achieve high performance in the implementation, a low overhead run-to-completion model is implemented and devices are accessed via polling to eliminate the overhead of context switches in the interrupt processing. The Linux kernel is bypassed and dedicated cores are used on Intel® i7 processors to directly process packets as they arrive from the NIC.

To run the Intel® DPDK application, the igb_uio module was first loaded into a running kernel. Binding the igb_uio driver to the device interface allowed the DPDK application to assume control of the device from the Linux kernel and drive the device in poll mode. The reservation of hugepages provided a large memory pool allocation for packet buffers in the DPDK application. By using these hugepages, it reduced the number of pages to be managed and increased the Translation Lookaside Buffer (TLB) hit rate. Hence, performance improved.

Figure 2 shows the block diagram of our implementation of KVS with Intel® DPDK. We start with the creation of an Environment Abstraction Layer (EAL) whereby the application first links to the libraries, and the device port initializes with the configuration parameters. Safe lockless Tx/Rx queues are also initialized and implemented instead of using spinlocks to avoid unnecessary wait time. Fixed size buffers are pre-allocated and created in the memory buffer pools of hugepage memory space, which reduces the time operating systems need to allocate and de-allocate buffers.

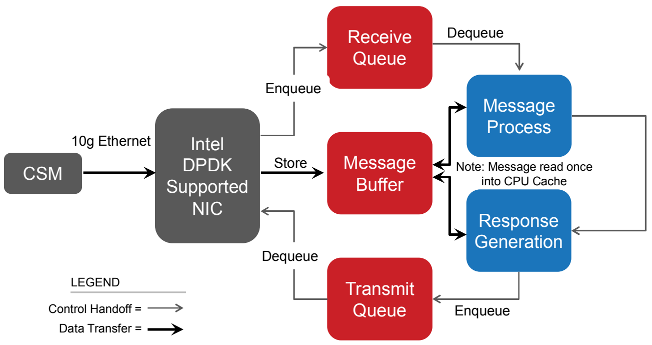


Figure 2: KVS with DPDK

After the initialization process is complete, the core assigned to the port starts polling the device. The program implements a receive method with the policy to retrieve as many received packets as possible. By calling the API function, the program loops and parses the Rx ring to receive queues that retrieve the packets with OCSMs, up to 32 burst packets in the current implementation. This approach of immediately processing received packets in fast bursts avoids the overhead of unnecessary packet enqueue and dequeue operations.

C. *Search using FPGA*

Rather than using a standard NIC to forward packets to software, an RTL implementation of the packet processing datapath and KVS is implemented in FPGA logic as shown in Figure 3. For this implementation, we use a Nallatech P385 board [P385] with an Altera Stratix V A7 FPGA. A block diagram of the Nallatech P385 board is shown in Figure 3.

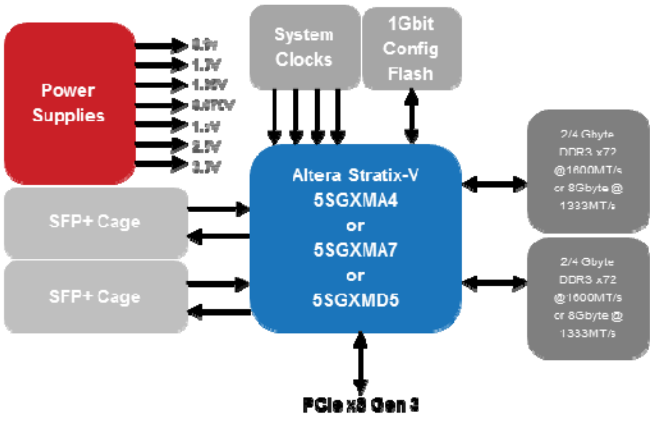

Power Supplies
1.2V
2.5V
3.3V
System Clocks
1Gbit Config Flash
Altera Stratix-V 5SGXMA4 or 5SGXMA7 or 5SGXMD5
SFP+ Cage
SFP+ Cage
2/4 Gbyte DDR3 x72 @1600MT/s or 8Gbyte @ 1333MT/s
2/4 Gbyte DDR3 x72 @1600MT/s or 8Gbyte @ 1333MT/s
PCIe x8 Gen 3

Figure 3: Nallatech P385 card with SFP+ ports and Stratix V A7 FPGA used for RTL implementation of KVS

The FPGA implementation of KVS uses the 2nd Generation Exact Match Search Engine (EMSE2) core [EMSE2] that emulates the function of a Binary Content Addressable Memory (BCAMs). CAMs have long been used by hardware designers to implement high-performance network lookup functions. BCAMs perform associative lookups from a fixed key. Ternary CAMs (TCAMs) have keys that can include wildcard (don't care) components in addition to fixed content. Traditional content addressable memories are implemented with a special type of associative memory that takes a key as an input and then compares the value of the key to the elements stored in the table. Unlike a traditional RAM, the address (index) where the key was found is the output instead of the input.

The EMSE2 core uses RAM combined with logic to emulate the function of a BCAM. As a result, the circuit requires less power than a brute-force CAM and is portable, allowing its operation as a soft core in FPGA logic devices.

Most CAMs require a secondary lookup after the search to perform a direct lookup of the index to a corresponding value. The EMSE2 includes this secondary lookup feature as a part of the search. A value can be associated with each key at the time it is inserted into the table. This value is returned whenever a search for a key is performed.

The EMSE2 core has two variations: an on-chip memory for all data accesses and a fully off-chip version. On-chip memory is best suited for small tables that require the fastest lookup rates and lowest latencies. A fully off-chip version of the EMSE2 core uses DDR3 memory to store up to 48M key-value pairs. The depth of the large table is only limited by the size of the off-chip memory.

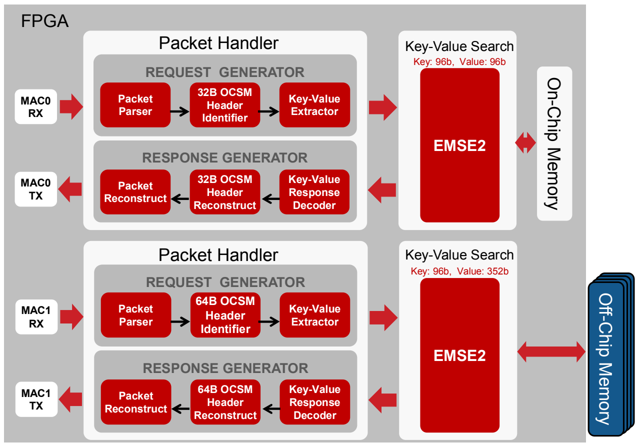


Figure 4: RTL datapath of KVS on Nallatech P385 expansion card

As shown in Figure 4, the two versions of the EMSE2 were connected to both on-chip and off-chip memories and to packet handlers that read data from the receive (RX) Ethernet mac, parse the packet, extract the key, then perform a lookup using the EMSE2. The resulting values are encapsulated into a response packet and sent to the transmit (TX) Ethernet mac.

IV. Scale Out of KVS for the Data Center

Several important high-level data center applications require scale-out workloads and benefit from the features supported by an FPGA-based KVS. In this section, we provide a few examples of applications that need large and fast KVS tables and describe how scale-out is implemented using traffic classifiers in Top-of-Rack Ethernet switches.

A. Scale Out Applications

Data centers hosting mobile applications can use a KVS to track customers and device locations [RoadID]. With over a billion mobile customers [Mobile Users Stats] currently active and more expected [internet.org], cloud providers need database services that scale up to support the increasing throughput generated by applications running on these mobile devices.

Mobile customers expect their service providers to allow them to access their data from multiple devices. KVS allows application developers to easily share data across multiple devices. Likewise, data acquisition systems, sensors, and Internet-attached devices that collect and analyze large volumes of data can benefit from the use of an underlying high-throughput KVS.

Hardware-accelerated equity, options, and futures trading exchanges benefit from the low latency and scalability of an FPGA-accelerated KVS. Exchange matching engines maintain order books to track the best bid and ask price for the underlying instruments. By distributing the content of the order book over multiple network-attached KVS instances, exchanges can scale up the number of symbols that can be traded and monitor all of the open orders in the marketplace while clients trading rapidly insert, delete, and modify orders. Distributed KVS enables scalability and provides redundancy for the data tracked by the exchange.

V. KVS over Ethernet

A. Scale Out over Ethernet

To search for data in a KVS table, a client machine sends a key in a packet and receives a response containing the corresponding value. The KVS can be provisioned to allocate tables across multiple types of KVS lookup engines. The decision where to forward the data can be assisted by a traffic classifier in the Ethernet switch. Higher throughput is achieved by replicating tables in the KVS and load balancing traffic across active tables. Redundancy is achieved by sending store and search requests to parallel copies of the KVS. Larger tables are implemented by distributing the key/values over multiple servers so that each server only stores a fraction of the table.

B. Rack-Scale KVS

As shown in Figure 5, KVS services can be deployed at rack scale using multiple KVS lookup engines and a Top-of-Rack (ToR) traffic classifier. Data from client machines elsewhere in the data center are received over 40 GE or 100 GE links and then forwarded to KVS servers within the rack via short SFP+ and QSFP+ direct attach cables. By implementing rack-scale services, entire database systems can be quickly deployed. The racks only need to be unloaded, powered, connected to the network, and provisioned.

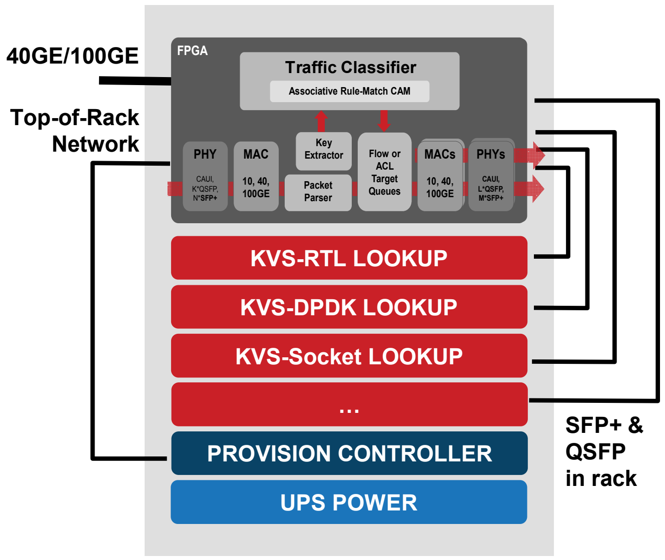


Figure 5: Rack with traffic classifier and KVS servers

C. Traffic Classifier

We implemented a fast traffic classifier on Altera's 100 GE FPGA evaluation platform [ALTERA100G]. The classifier selectively forwards packets from a 40 GE or 100 GE input port to multiple 10 GE-connected endpoints within a rack. Under software control, rules were programmed in the traffic classifier module to load the KVS servers. Like N-Tuple matching with a Software Defined Networking (SDN) switch, the Gateware Defined Networking (GDN) traffic classifier parses multiple fields to select the destination of packets sent to the ToR with KVS commands. Load balancing is supported by classifying packets based on the source address of the packet.

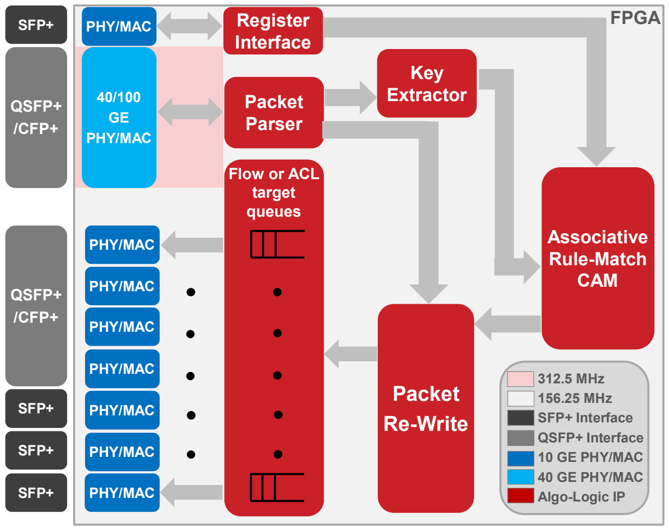


Figure 6: Datapath for 40/100 GE Traffic Classifier

A detailed block diagram of the traffic classifier module is shown in Figure 6. One of the 10 GE ports is connected to the host to program the rules in the CAM. Data enters the FPGA through the QSFP+ port, passes through the 40 GE PHY, and is forwarded to the MAC. The *packet parser* then sends the header to the key extractor and the payload to the *packet re-write* module. The *key extractor* constructs a key from the header of the packet. This key is then matched with the entries in the CAM. The matching rule determines how the packet header is remapped and forwarded to specific target queues. These queues are in turn serviced by the transmit side of the MACs and forwarded to the corresponding PHYs.

For packets that arrive on a 40 GE interface, 256 bits of data are processed in each cycle of the datapath. The *key extractor* module generates an 80 to 640 bit key by extracting the required fields from the header of the incoming packet. The header fields are available in the first 48 bytes after the start of packet is received or in two 32 bytes words on the datapath. Simultaneously, the *packet re-write* module buffers the payload in the payload FIFO to be appended to the outgoing packet.

D. *KVS-RTL Implementation*

The RTL datapath, as shown in Figure 4, instantiates Algo-Logic's Ethernet 10 GE MAC, which interfaces with the p*acket parser* module. This module filters the UDP traffic and forwards the payload to the *header identifier* module that identifies the type (Update/Search) of KVS messages. The key-value extractor is then responsible for extracting the 32B or 64B key-value pairs that are stored in tables maintained using either the on-chip or off-chip memory. To support a wide value and large number of entries, we use the off-chip memory to store the 64B format. The EMSE2 IP enables the access to different tables.

The depth of the on-chip table is limited by the available on-chip memory of 6.25 MB on a Stratix V GXMA7 [P385] device. With 15% of ALMs, 13% of Registers, and below 75% of on-chip memory usage, the current on-chip table is 48K deep and the off-chip table is 12M deep. The off-chip table can be scaled up to 48M entries with the available 8GB of DDR3.

On a 10 GE link, the interface between the MAC and *packet parser* module operates at 156.25 MHz. The EMSE2 IP, which enables access to the on-chip table, can process messages every clock cycle. However, maximum throughput of 10GE with 32B messages limits the maximum search rate to 40 MSPS. The performance of the EMSE2 IP, which enables access to the off-chip table, is determined by the number of accesses supported by the DDR3 memory controller for which the estimated theoretical upper limit is 30 MSPS. However, the number of 64B messages received on a 10 GE is 20 MSPS, which is the maximum throughput that can be achieved from an off-chip table.

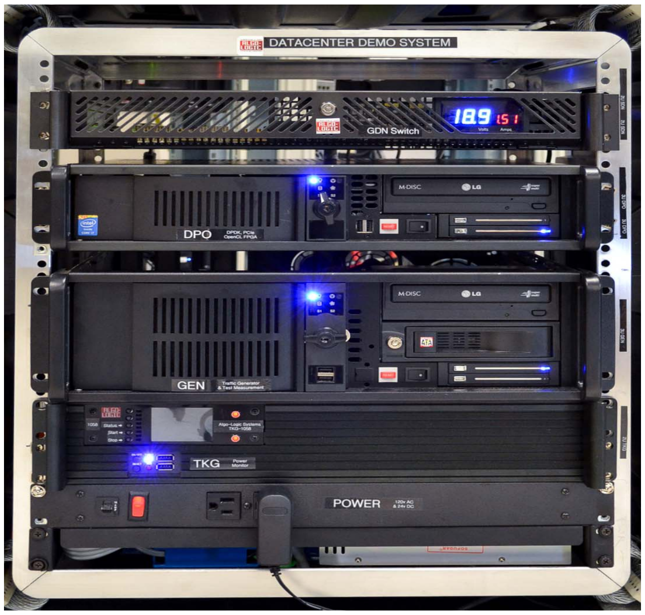


Figure 7: System used to generate traffic, implement KVS, and measure latency, throughput, and power.

E. *System Implementation*

A photograph of our system used for testing KVS with the three implementations is shown in Figure 7. The top rack contains the ToR traffic classifier that implements the associative rule-match CAM module, which filters the packet for software, DPDK, and RTL KVS using the rules in the traffic classifier. The second rack contains the lookup server containing the i7 CPU, Intel 10 GE NIC, and the FPGA card. The third server contains the traffic generator that includes a 40 GE NIC used to run TCP-Replay. The fourth server records power consumption. Finally, the bottom of the rack contains the power supplies that source the 24V DC.

VI. BENCHMARKS

Each of the software, DPDK, and RTL implementations of KVS using the on-chip table were evaluated using a series of tests to determine latency, throughput, and power usage. As discussed in [BERK SIG], the 99.8% of the KVS workload is comprised of lookup requests whereas update and delete are used infrequently. Thus, we use the KVS lookup commands to benchmark latency and throughput.

A. *Latency Measurement Mechanism*

The traffic classifier module computes the latency for each KVS lookup command for each packet by using the difference in time from when the packet was seen at the egress to the time that it is returned to the ingress.

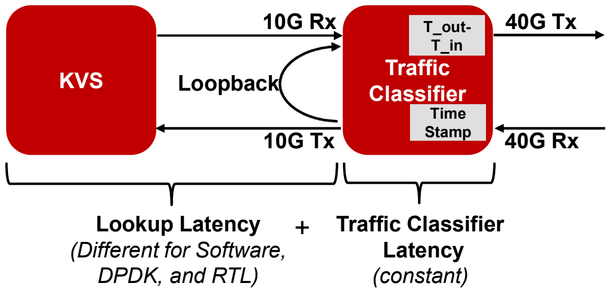

KVS
10G Rx
Loopback
10G Tx
T_out-T_in
Traffic Classifier
Time Stamp
40G Tx
40G Rx
Lookup Latency (Different for Software, DPDK, and RTL)
+
Traffic Classifier Latency (constant)

Figure 8: Setup for latency measurement

As shown in Figure 8, we compute the latency of the traffic classifier by using a loopback connector on the 10 GE port. We then subtract the constant latency of the traffic classifier from that of the lookup latency. Just before each packet is written from the MAC to a 10 GE, the classifier records a timestamp in the packet. A 24-bit sequence number is written as the first three bytes in each datagram within the 6-byte pad that precedes the OCSM. The time between the request and response packets, measured in the traffic classifier's clock cycles, precisely tracks the latency of each lookup. A distribution graph was generated to show the percentage of packets that experienced latency for each of the KVS implementations.

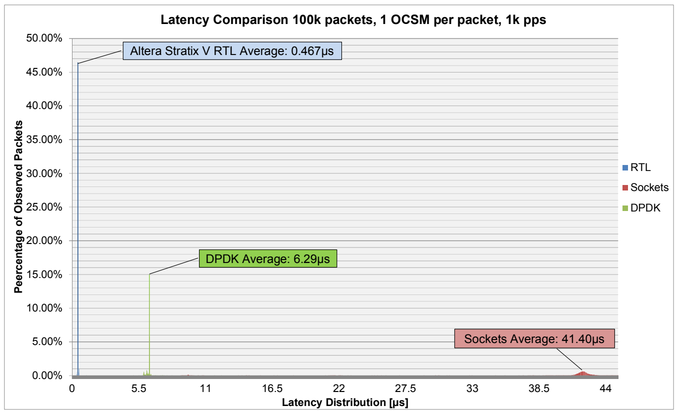


Figure 9: Latency distribution for FPGA, DPDK, and socket implementations of KVS

As shown in Figure 9, the RTL implementation of the KVS using an Altera Stratix V FPGA achieved the lowest latency of 0.467 µs. Because the number of clock cycles required to process the search is engineered into the logic as a constant, the search achieves jitter-free operation, thus appearing as an impulse function on the graph.

The DPDK implementation had the next lowest latency of 6.43 µs. By dedicating a CPU core to spin-wait on incoming packets, the overhead of the Linux kernel was avoided.

Finally, the traditional software socket implementation had the worst average latency of 41.54 µs. The overhead of forwarding packets through the kernel and using software to sequentially process the packets was approximately 88 times slower than the results achieved with the native RTL implementation.

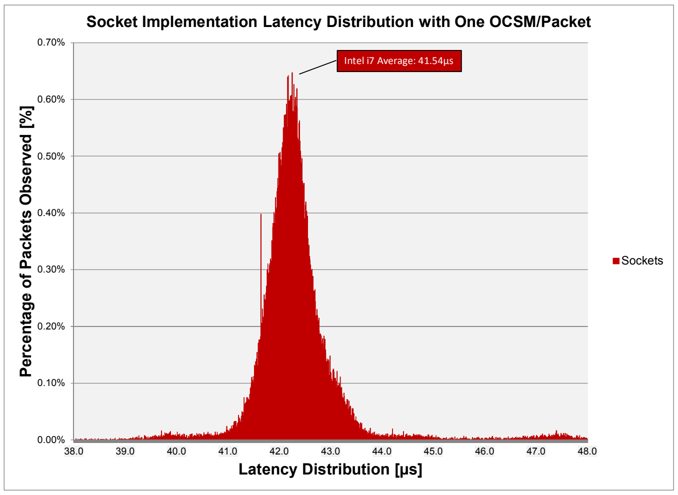


Figure 10: Detail of latency distribution for software socket implementation of KVS. Note wide variation in latency (jitter).

Details of the latency distribution of the socket implementation are shown in the graph in Figure 10. We see that software has a wide variation of approximately 10 µs in the time required to process packets with a standard deviation of approximately 2 uS.

B. *Throughput Measurement Mechanism*

The throughput test performed a series of sub-tests that each by send requests at a specific rate. The setup is as shown in Figure 11. The number of OCSMs sent and received was compared. The throughput was determined as the maximum request rate that results in all sent messages processed properly. The maximum throughput of 40 MSPS from the table stored in on-chip memory is limited by the number of 32B messages received on a 10 GE link. The maximum throughput of 20 MSPS from the table stored in off-chip memory is limited by the number of 64B messages received on a 10 GE link.

The test generator created a packet capture file (PCAP) that was replayed by the traffic generator. Before and after the traffic was replayed, the packet counters on the traffic classifier between the traffic generator and the design under test were retrieved. The difference between the counters before and after determined the number of packets that each implementation received and the number that were sent.

The three implementations were tested using two separate machines. One machine was configured as the traffic generator and was used to run the test suite. The other machine held the hardware for the KVS implementations. Three separate 10 GE links connected the two machines, one for each implementation. Only one implementation was active during testing to increase the accuracy of the results.

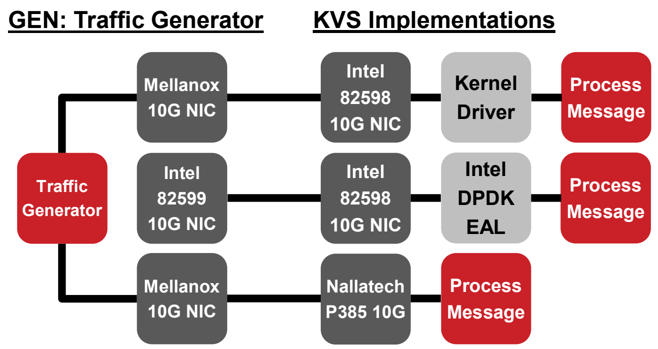


Figure 11: Traffic Generation for Testing KVS Implementations

The lines in graph in Figure 12 show the percent of messages processed as a function of the traffic sent to the system. All three implementations (the RTL circuit, DPDK software, and software sockets) initially processed 100% of the messages received. However, as the traffic load increases beyond a rate of several million messages per second, the fraction of packets processed decreased.

First, the socket implementation became unable to sustain full throughput. Next, the DPDK implementation became unable to maintain full throughput. However, the RTL implementation was able to maintain full throughput up through the maximum speed of the traffic generator.

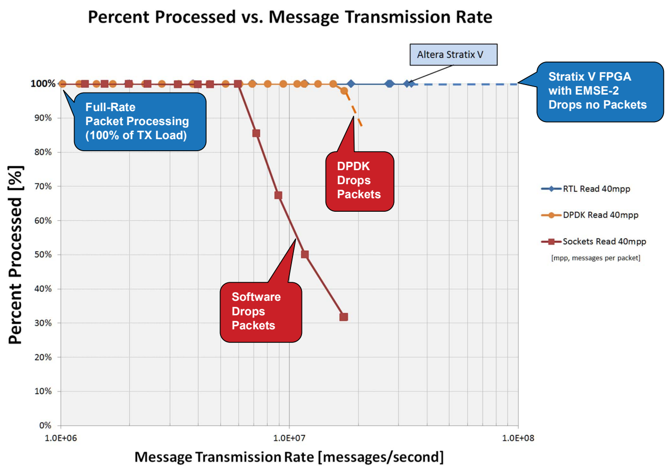


Figure 12: Throughput comparison of Software, DPDK and RTL implementation of KVS

C. Power Measurement Mechanism

Power usage wsa determined by taking the system's rate of power consumption and dividing it by the system's maximum throughput with the result measured in units of Joules/OCSM. To maximize power efficiency, the search server ran from DC power.

We measured the power consumed by the system comprised of the complete Rack-PC, including both an Intel Core i7 4770k CPU and an Altera Stratix V A7 FPGA on a Nallatech P385 PCI-express board. We used Hall-effect current sensors as well as a voltage monitor to compute the power consumed during computation either in hardware or software.

To record the data over time, we monitored the current and voltage with a rack-mounted measurement system [BlackDiamond]. The graph in Figure 13 shows the current as a function of time in a one-minute duration when the DPDK server was running and processing packets. The y-axis indicates the current (I) measured in amps (A) for a given time within that minute. Each interval on the x-axis represents 0.1 minutes or 6 seconds. The power supply provided a constant voltage of 26.0V.

The current initially stayed at 1.74 A. When the DPDK server was launched, the device was initialized and the current increased to a range between 2.7 and 2.8 amps as shown in Figure 13. Then the device spent approximately three seconds checking the link status. Once the link was established, the DPDK server started polling for packets and the current increased to 3.19 A.

When the packets arrived, the DPDK server started receiving, processing, and transmitting in a burst-oriented mode, which actually caused the CPU to consume slightly less energy than it consumed when spin waiting.

While receiving, processing, and transmitting the OCSM, the marginal energy consumption was 2.83 µJoules per OCSM and the total energy consumption was 6.59 µJoules per OCSM.

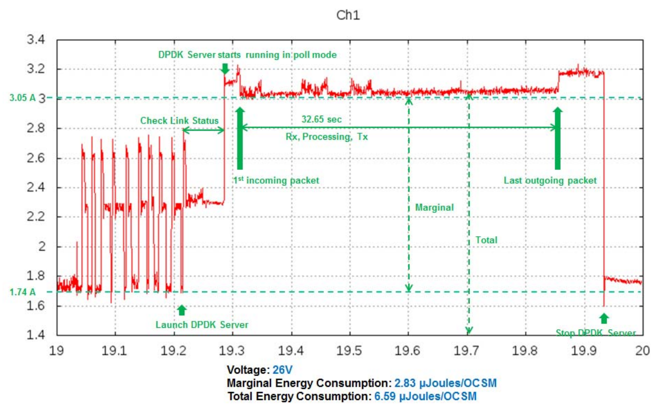


Figure 13: Time history of power (Amps*26V) consumed by CPU polling while running the DPDK version of KVS as a service

In one experiment, we processed 40 OCSMs per packet while in the other experiment each OCSM was carried in its own packet. We considered both the total energy (energy to operate the server + application) as well as the marginal energy (energy to solely perform the KVS lookup above and beyond the base level of the server). A summary of the results is show in Figure 14 and Figure 15.

10M packets	40 CSMs/packet	
	Marginal Energy (µJoules/OCSM)	Total Energy (µJoules/OCSM)
Socket	3.53	11.07
DPDK	2.83	6.59
RTL	0.12	0.52

Figure 14: Energy consumed by each implementation of KVS: Software sockets, software DPDK, and RTL running on the FPGA.

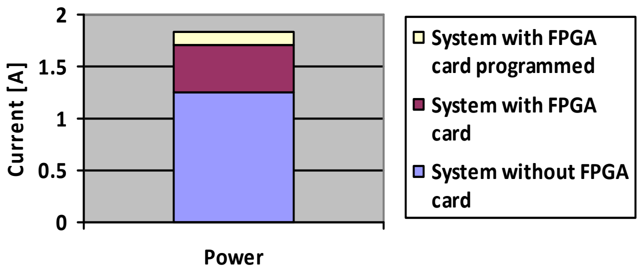


Figure 15: Base power (Amps * 26V) of i7 server while idle, with FPGA, and with marginal power to run KVS as service

The RTL implementation achieved, by far, the lowest energy consumption both in terms of marginal power and total power. In fact, the marginal power was so low that it was below the threshold at which the difference in system power could be measured while the packets were being processed. Since the operation was completely offloaded from the CPU, the computing system remained idle during the test.

D. Summary of Key Results

A summary of performance for socket, DPDK, and RTL is shown in Figure 16. As we can see, the RTL implementation achieved the highest performance. In comparison with DPDK, the latency and power consumption of RTL are 10x and 13x less than DPDK respectively, while the maximum throughput of RTL is 3.2x larger than DPDK. In addition, the RTL power consumption of 0.52 µJoules/CSM, is much lower than the 33 µJoules/CSM in another RTL implementation described in [Blott USENIX].

The socket implementation has the worst performance. The latency and power consumption of RTL is 88x and 21x less than the socket respectively, while the maximum throughput of RTL is 13x larger than the socket.

All Datapaths Summary	Latency (µseconds)	Tested Throughput (CSMs/sec)	Power (µJoules/CSM)
Sockets	41.54	4.0	11
DPDK	6.434	16	6.6
RTL	0.467	15	0.52

All Datapaths Summary	Latency (µseconds)	Maximum Throughput (CSMs/sec)	Power (µJoules/CSM)
GDN vs. Sockets	88x less	13x	21x less
GDN vs. DPDK	10x less	3.2x	13x less

Figure 16: Summary of performance metrics for implementation of KVS using software sockets, software DPDK, and RTL running on the FPGA

VII. Related Work

A small demonstration with open source example code showing a simple KVS as a service over the public web was demonstrated with [OpenKeyval]. Keys were mapped to values in a system that has minimal security and all data in a single namespace.

Memcached is the most commonly used implementation of KVS applications within current web infrastructure. Originally developed by Danga Interactive for LiveJournal, Memcached is now used as a service at YouTube, Zynga, Facebook, Twitter, Tumblr, and Wikipedia as well as being used as a part of the platform for the Google App Engine, AppScale, Microsoft Azure, and AWS [Memcached].

Most memcached implementations are deployed using x86 based servers that have limited performance scalability and high performance consumption. However, an FPGA-based memcached architecture can achieve two orders of magnitude of performance improvement over standard x86 based approaches. When x86 based implementations provided up to 1.4 MSPS with a latency of 200-400 us, a corresponding FPGA implementation was shown to provide 13.02 MSPS with a worst-case latency of 3.5 to 4.5 us [Blott USENIX].

DynamoDB, available as a managed service from Amazon [DynamoDB], claims to provide single-digit millisecond latency (corresponding to several thousand microseconds) for KVS operations and a 99.9% bound on the worst-case write latency no greater than a few hundred milliseconds. Data are replicated to improve reliability with no strong guarantee for write consistency [Dynamo]. The service has no fixed schema but instead allows each item to have a number of attributes that can be strings, numbers, binary data, or sets. Atomic counters allow for numbers to be incremented or decremented with a single API call [DDB-Dev-Guide]. The service scales to run on a varied number of machines ranging from one to over hundreds of servers.

CONCLUSIONS

A high performance Key Value Store (KVS) with an implementation in logic that has properties optimal for deployment in data centers is presented. The KVS uses the new OCSM message format to efficiently pack data in compact binary messages in standard Internet Protocol (IP) packets transmitted over Ethernet.

Three implementations of KVS were designed, verified, tested in hardware, and documented. First, a traditional software socket implementation that listened on the network was coded in user space. Second, optimized software was implemented using Intel's Data Plane Development Kit (DPDK) that runs on a dedicated processor core on an Intel i7 CPU. Third, an RTL-based KVS was implemented on an Altera Stratix V FPGA.

To test the resulting implementations, a rack system was assembled that contained Altera's 100G Development board as a ToR traffic classifier, a 4th-generation Intel Core i7 CPU system equipped with a dual-port Intel 10 GE NIC, and an Altera Stratix V FPGA card. These hardware components were used to implement the three variations of the KVS algorithm in sockets, DPDK, and RTL. A traffic generator with a 40 GE NIC card was used to reply to a burst of packets with OCSMs and a rack-mount measurement system was used to record power and energy consumption.

Measurements were obtained for each of the three implementations for throughput, latency, and power. In terms of throughput, captured packets were replayed to saturate the ports of the search implementations. Both the socket implementation and DPDK dropped packets as the traffic load was increased, but the RTL implementation was able to respond to 100% of the traffic load up to the maximum speed of the traffic generator.

In terms of latency, the RTL implementation of the KVS using an Altera Stratix V FPGA achieved the lowest latency of 0.467µs, the DPDK implementation had the next lowest latency of 6.43µs, and the traditional software socket implementation had the worst average latency of 41.54µs. The overhead of forwarding packets through the kernel and using software to sequentially process the packets was approximately 88 times slower than the results achieved with the native RTL implementation. As compared to the fastest known software approach, the FPGA KVS was measured to process messages 10x faster while using 13x less energy than kernel bypass software.

ACKNOWLEDGEMENTS

The authors would like to thank members of the Algo-Logic team for help in developing and testing the KVS hardware and software. We appreciate the comments from the members of the Hot Interconnects technical program committee. We also greatly appreciate the help of Emily Iwankovitsch who edited multiple drafts of this document.

REFERENCES

[Aerospike] Aerospike, "The first, flash-optimized in-memory, NoSQL database", Slides available at http://insideanalysis.com/wp-content/uploads/2014/08/Aerospike_BDE_June132014.pdf?iframe=true &width=1080&height=784

[ALTERA100G] http://www.altera.com/products/ip/iup/ethernet/m-alt-40-100gb-ethernet.html

[BERK SIG] Berk Atikoglu, et al, "Workload analysis of a large-scale key-value store", *12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (SIGMETRICS '12). ACM, New York, NY, USA, 53-64

[BlackDiamond] http://algo-logic.com/bdr, May 2015

[Blott FPL] Zsolt István, Gustavo Alonso, Michaela Blott, Kees A. Vissers: A flexible hash table design for 10GBPS key-value stores on FPGAS. FPL 2013: 1-8

[Blott USENIX] Blott, Michaela, et al. "Achieving 10gbps line-rate key-value stores with fpgas." Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing. USENIX, 2013.

[DDB-Dev-Guide] Amazon Dynamo DB Developer Guide, API Version 2012-08-10, http://awsdocs.s3.amazonaws.com/dynamodb/latest/dynamodb-dg.pdf

[DPDK] Intel® Data Plane Development Kit - Programmer's Guide, http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-dpdk-programmers-guide.pdf, October 2013

[DynamoDB] AWS | Amazon DynamoDB - NoSQL Cloud Database Service - http://aws.amazon.com/dynamodb/, April 16, 2014

[Dynamo] Dynamo: Amazon's Highly Available Key-value Store, by Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, Symposium on Operating System Principles (SOSP) 2007, http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[EMSE2] http://algo-logic.com/emse2

[Ferdman SIGARCH]: Ferdman, M., et al, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware", SIGARCH Comput. Archit. News 40, 1 (Mar. 2012), 37–48

[internet.org] internet.org by Facebook available at https://www.internet.org/about

[Intel82598] Intel® 10GE Controller Datasheet (82598), http://www.intel.com/content/www/us/en/ethernet-controllers/82598-10-GE-controller-datasheet.html

[McKinsey] James Manyika, et al, "Big data: The next frontier for innovation, competition, and productivity (May 2011)", http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation

[Memcached] http://en.wikipedia.org/wiki/Memcached, April 16, 2014

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

[Memcache-FB] Scaling Memcache at Facebook - USENIX 2013 https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf

[OpenKeyval] http://openkeyval.org/

[Mobile User Stats] http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics

[P385] http://www.nallatech.com/solutions/fpga-accelerated-computing/pcie-accelerator-cards/

[Pokorny IJWIS] Jaroslav Pokorny (2013), "NoSQL databases: a step to database scalability in web environment", International Journal of Web Information Systems, Vol. 9 Iss 1 pp. 69 – 82

[RoadID] http://www.roadid.com/eCrumbs