Key-Value-Links: A New Data Model for Developing Efficient RDMA-Based In-Memory Stores

Hai Duc Nguyen, The De Vu, Duc Hieu Nguyen, Minh Duc Le, Tien Hai Ho and Tran Vu Pham
Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam
E-mail: ptvu@hcmut.edu.vn

Keywords: in-memory stores, key-value, RDMA, InfiniBand

Received: March 24, 2017

This paper proposes a new data model, named Key-Value-Links (KVL), to help in-memory store utilizes RDMA efficiently. The KVL data model is essentially a key-value model with several extensions. This model organizes data as a network of items in which items are connected to each other through links. Each link is a pointer to the address of linked item and is embedded into the item establishing this link. Organizing datasets using the KVL model enables applications making use RDMA-Reads to directly fetch items at the server at very high speed. Since link chasing bypasses the CPU at the server side, this operation allows the client to read items at extremely low latency and reduces much workload at data nodes. Furthermore, our model well fits many real-life applications ranging from graph exploration and map matching to dynamic web page creation. We also developed an in-memory store utilizing the KVL model named KELI. The results of experiments on real-life workload indicate that KELI, without being applied much optimization, easily outperform Memcached, a popular in-memory key-value store, in many cases.

Povzetek: Predlagan je nov podatkovni model, imenovan Key-Value-Links (povezave ključnih vrednosti).

1 Introduction

In-memory stores have flourished in recent years owing to the urgent needs of fast processing and decreasing DRAM prices. Many system designers have either used main memory as a primary data store [17] or as a cache to reduce the latency of accessing hot or latency-sensitive items [2]. Being moved to main memory enables data to be accessed at very low latency because it removes the overhead of disk and flash. But it does not mean I/O overhead is absolutely eliminated. Because of DRAM's low capacity, in-memory stores often deploys across multiple data nodes making network I/O become a potential source of overhead. Indeed, the traditional TCP/IP networks have shown many disadvantages in supporting fast data transmission. For example, MemC3, a state-of-the-art in-memory store, runs seven times better on a single machine than in a client-server setup using TCP/IP [9, 8].

To solve this problem, several data centers started looking for alternative solutions. Among those, Remote Direct Memory Access (RDMA) is appeared to be the most promising candidate. RDMA allows applications to directly read from and write to remote memory without involving the Operating System at any host. This ability helps RDMA achieve low latency and high throughput data transmission because it bypasses the overhead of complex protocol stacks, avoids buffer copying, and reduces CPU overhead. Despite attractive features, RDMA has not been widely used in data centers due to the high prices of its supporting NICs. In recent years, however, the prices of RDMA-enabled NICs have dramatically dropped and become compatible with that of traditional Ethernet NICs. For examples, A 40 Gbps InfiniBand RDMA-capable NIC costs around $500, while the prices of a 10GB Ethernet NICs may be up to $800 [15]. New standards such as iWARP and RoCE also support RDMA allowing data centers to utilize RDMA with reasonable cost.

Within this trend, there are many studies have started to leverage RDMA technology to build ultra-low latency in-memory store. Those works indicate that much of effort have to be spent in order to maximize the benefits of using this technology for in-memory systems. Works to be done including reduce NIC's cache miss rate [8], minimizing the number of RDMA operations per requests [15, 8], and optimizing hash table organization [8, 15, 12], etc. In spite of implementation differences, most of existing RDMA-based in-memory stores are constructed according to key-value model since this model is very simple and well fit large and unstructured datasets.

The key-value model, however, has its own drawbacks. The most noticeable one is performance. Traditionally, every put and get operation involves to hash table lookup to identify the existence of items. This makes the hash table become the hotspot of data access and it is not surprising that most of the key-value stores spend much of effort tuning their hashing mechanism [15, 8, 9]. Real-life workloads indicate that key-value items are typically small [3] so employing key-value model could be easily suffered from low

network utilization. Furthermore, using key-value model often has applications to divide its requests into multiple small item lookups. Those lookups often have to be executed sequentially due to data dependency. This causes the hash table lookup overhead and low bandwidth utilization of multiple lookups to accumulate and reasonably prolong the latency of the original request. According to [17], Facebook creates about 130 internal requests in average for generating the HTML for a page. Similarly, Amazon requires about 100-200 requests to create HTML part for each page [7]. With those workloads, in-memory stores have to react very quickly to each request to guarantee desired performance. In the future, as the amount of data and workload keeps increasing rapidly, it is difficult for the in-memory stores to maintain its performance without changing request processing mechanisms.

In this paper, we introduce a novel data model named Key-Value-Links (KVL) to enable in-memory stores to exploit RDMA efficiently to deliver ultra-low latency data services. Essentially, the KVL model is a variant of the Key-Value model that maintains links between items to exploit the data dependency between them to accelerate data retrieval. A link contains the information about the (physical) location and the size of referred item so applications could utilize RDMA Reads to directly fetch the item without invoking expensive item lookups. This design bypasses the hash table and efficiently utilizes the network as we need only one RDMA Read for reading an item. As a result, getting desired items by chasing their links reduces the cumulative latency of processing multiple item requests significantly. We also introduce KELI (KEy-value-with-Links In-memory), an in-memory cache that employing KVL data model, and compare it with an in-memory key-value cache (Memcached) to reveal the performance benefits of utilizing the KVL model over RDMA-capable networks.

The following section briefly introduces RDMA technologies and recent works in developing in-memory stores using RDMA. Section 3 discusses the KVL model in detail. Several classes of applications which could utilize the model efficiently are listed in Section 4. The section 5 discusses the design of KELI. We conduct several experiments on real-life data to evaluate the efficiency of using KELI and report their results in Section 6. Finally, we conclude the paper in Section 7.

2 Background and related work

2.1 Remote direct memory access

Remote Direct Memory Access (RDMA) allows remote computers to directly read memory regions on local memory without interfering its CPU. This allows zero-copy data transfers and saving computing resource. Furthermore, RDMA-enabled NICs provide kernel bypass for all communications and reliable delivery to applications. These make the typical latency of interconnects supporting RDMA such as InfiniBand, RoCE and iWARP about 10x faster than traditional Ethernet [12]. RDMA-enabled NICs is originally designed for High-performance Computing centers but due to decreasing in hardware prices, their presence in data centers is increasing [15]. The introduction of RoCE and iWARP, which lets RDMA to be performed over traditional network architecture, even makes RDMA more popular.

Applications utilize RDMA-enabled NICs through Verb API. There are several types of verbs but the most common are RDMA Read, RDMA Write, Send and Receive. These verbs could be grouped into two types of semantics: channel semantics and memory semantics. Send and Receive have channel semantics: to send a message, the sender posts a Send description to put the message content to a remote memory location specified by a pre-posted Receive description at the receiver side. Send and Receive are two-sided verbs as the communication involves the CPU of both end points. RDMA Read and RDMA Write have memory semantics: they operate directly upon the remote memory regions. Both of them are one-sided as the remote CPU does not aware those operations. This reduces not only the overhead of RDMA operations but also the load of remote CPU. Therefore, utilizing one-sided RDMA verbs could achieve very low latency and high throughput.

2.2 In-Memory stores using RDMA

Attractive features of RDMA verbs have exposed many studies of utilizing RDMA technology to build high-performance in-memory stores. As communication is the major source of overhead, previous work tries to replace traditional data transfer techniques by RDMA operations to effectively reduce the overall latency. For examples, Jithin Jose et al. [11] improves Memcached performance by the factor of four by just making it RDMA capable. In the later work [10], the research group uses a hybrid approach that utilizes both Reliable Connection (RC) and Unreliable Connection (UC) transport and transparently switches between them to further improve the performance by the factor of 12.

Apart from communication, recent studies have started to apply multiple optimization techniques on other parts of the system to reach even better performance. HERD [12, 13], makes heavy changes ranging from reducing network round trips and reorganizing data distribution to optimizing PICe transactions. It even sacrifices the reliability to maximize the performance. RDMA is also combined with other technologies to develop complicated in-memory stores. DrTM [23] and DrTM+R [6] are two fast in-memory transaction systems utilizing both RDMA and hardware transactional memory (HTM).

Different from traditional designs, Pilaf [15], FaRM [8] and HydraDB [22] let applications process request by themselves through RDMA Reads. In these systems, the server makes data visible to clients so the client could use RDMA Reads to access hash table and items at the re-

mote server as if they are on its own local memory. This approach bypasses many sources of overhead and reduces load at data servers but there are shortcomings preventing those systems from maximizing the potential of RDMA Read. For examples, Pilaf clients have to carry out multiple RDMA Reads per request. FaRM often performs RDMA Reads to get memory blocks which are much larger than the actual size of needed item. Also, bypassing remote CPU makes it unaware about application behaviors which are useful for tracking popular items.

Despite differences in implementation, all studies mentioned above employ the key-value model. Although this model is quite simple and easy to implement, the lack of the ability to represent complex data force applications using this model to generate a lot of item lookups for each data request. This disadvantage makes the key-value model sensitive to latency. This is the motivation for us to develop Key-Value-Links model to solve this issue.

3 Key-Value-Link data model

3.1 Example

Before describing the Key-Value-Links (KVL) model in detail, let us first show how it "looks and feels" through an example of using this model to represent a real-life dataset and handle data requests. Suppose we have a database storing the information about students, professors, and departments in a university. Figure 1 illustrates how the KVL is used to organize this database. In this representation, each entity (i.e. student, professor, and department) is a key-value item. The key is unique and is used to identify the item. The database has five entities: two students *"stu001"* and *"stu002"* under supervision of two professors *"prof001"* and *"prof002"* working in the department *''dpcs"*. The value of each of those items contains multiple attributes representing information associated with the item. The item *"stu002"*, for instance, has three attributes in its value. The first one (e.g. *"name : DEF"*) shows the student's name while the other are links indicating his supervisor and mentor. Those links do not provide information about those people but instead point to items storing information about them. In implementation, those links could be represented as pointers which let applications directly access linked item without sending a request to the data server.

Storing links to other items inside the item's value makes it easier to reason useful information which requires us to combine the data from multiple sources. In the university database, for example, suppose that user wants to know whether two students *"stu001"* and *"stu002"* are under the supervision of professors working in the same department. To answer this question, we must first get access to the two student items using their keys. After that, we travel across the supervisor link of those items to obtain the information about the supervisors. We then use department links to go to their departments. Finally, we check if those departments are the same to provide the final answer to the question.

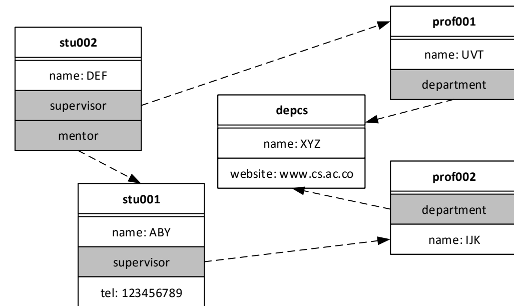


Figure 1: An example of representing a dataset from a university using the KVL data model. The key of each item is shown in bold text, the links are shown in shadow frames, and the arrows used to represent the link from one item to another.

If this database is represented by the relational model, answering this question requires us to perform multiple cumbersome joins. Because such operation consumes a lot of time and resource, using a relational database in such case does not guarantee an acceptable performance. If we use traditional key-value model, the applications must decompose the request into multiple item lookups. Since most of the lookups depend on the results of the previous ones, applications have to perform them sequentially. If the number of lookups is large, the cumulative latency, which is calculated by adding up the latency of each item lookups, will become very high and reasonably hurts the overall latency of the original request. Most of the item lookups could be replaced by one link chasing if we use KVL model. As we will show in the next subsection, the former operation is much more expensive than the latter so using key-value model also takes more time to process the request than applying the KVL model.

3.2 Data model

Generally, KVL is an enhanced version of the traditional key-value model. In this model, each item is a key-value pair connected to each other through links. Inside an item, the key is its identifier while the value describes its characteristics. There is no restriction on the size of either key or value. Different from some implementations of key-value model used in RAMCloud [17] or Memcached [2], KVL model cares about the structure of value. Particularly, the value is a set of attributes in $< K, V >$ format where K is the name of the attributes and V is its value. The value V could be either a block of bytes representing some kind of information defined by the user or a link to another item.

The concept of links is similar to that of pointers. Both of them let applications know the location of the resource but do not provide the information about its content and associated data. There are several benefits of this approach. Embedded an item into another one enlarges data size sig-

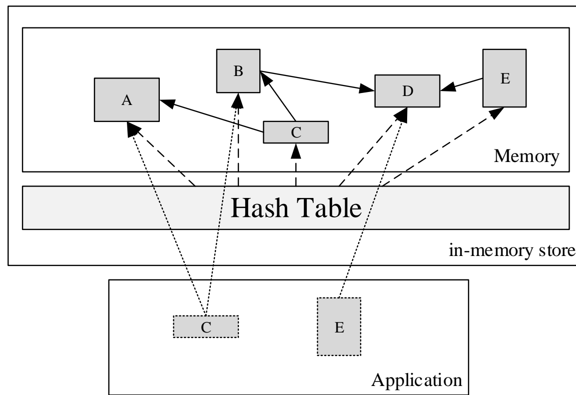


Figure 2: An example implementation of KVL model based on the organization of existing key-value stores.

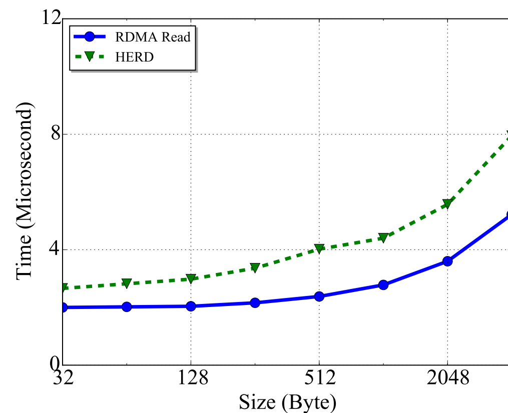


Figure 3: The latency of getting items with different sizes using RDMA Read and HERD.

nificantly which would reduce memory utilization as well as slow down data transmission. This also allows an item to have multiple copies which could be a nightmare for maintaining consistency. Furthermore, with the support from RDMA Reads, pointing to referred item through its address lets applications directly fetch the item without interfering the data server. Utilizing RDMA Reads helps fetching item at ultra-low latency as it bypasses many sources of overhead such as notifying remote CPU and hash table lookup. It also allows the system scale easily as the remote machine could save many CPU cycles for other tasks.

Figure 2 illustrates the organization of an in-memory store implementing the KVL model based on the fundamental structure of existing in-memory key-value stores. Basically, KVL model is also a key-value model so methods it uses to handle data are similar to those of key-value. In particular, the store constructs a hash table to keep track of items stored in the system. Putting a new item to and getting an existing one from the store requires the key of this item to be hashed to the hash table first to determine the proper action. Clearly, operations in a key-value store are all related to the hash table making it become the hotspot of the system.

The introduction of links leads to a new way to get data from in-memory stores called link chasing to reduce the load on the hash table. In this method, applications use links attached to items it has fetched previously to invoke RDMA Reads to directly retrieve the linked items from the in-memory stores without explicitly sending a get request. For examples, in the Figure 2, the application has performed two lookups to load the item B and item C from the server. It has two options to load item A. It could generate a get request containing the key of item A and send it to the server to have it search for this item. The another choice is to use RDMA Read to chase the link to item A which is integrated to the item C to read this item directly without asking for the server.

Figure 3 compares the latency of link chasing using RDMA Read with that of item lookup on different item sizes. The implementation of item lookup is based on the method used in HERD [12], which is one of the fastest in-memory key-value stores in the literature. It is clear that even though being heavily optimized by many techniques, item lookup still runs much slower than RDMA Read. This means if we could organize items needed by applications in a way such that from one item we could reach to other ones by just chasing links, the latency could be reduced up to 50%. Therefore, using KVL data model with good data schema design could significantly boost the system performance without spending much effort on optimizing the in-memory store implementation.

4 Applications

Apart from the simple university example in the previous section, we found that the KVL is also applicable to a wide range of applications. The followings are a few of them.

4.1 Graph exploration

Graph exploration is required by many data-intensive applications [16]. Graph traversal algorithms such as breadth-first search (BFS) and depth-first search (DFS) are used as basic components in various complicated algorithms which are used to solve problems in many fields including biology, communication, social network, etc. In the Big Data area, a graph could contain up to trillions of nodes and edges. Hence, traverse such large-scale graphs efficiently is critical.

A graph contains only nodes and edges but in real-world applications, both nodes and edges are associated with a lot of information. This makes representing the topology of the graph in the computer a nontrivial task especially in the case of large graphs which could expand over multiple data nodes. Modeling graphs using relational model or XML does not scale well. Plus, those tools do not ease graph traversals. Many state-of-the-art in-memory graph

databases are constructed upon key-value model [4, 21] due to its simplicity. However, deploying graph traversal algorithms using this model would lead to high cumulative latency.

The KVL model, on the other hand, is very similar to the concept of Graph database since this model itself is a network of items. In fact, it could be considered as a "lightweight" graph in which items are vertices and links are edges. We use the term "lightweight" because there are some limitations that prevent KVL from naturally representing complicated graphs. For examples, information cannot embed into links and a link must point to a physical address rather than abstract objects. Due to those shortcomings, using links to represent edges in complex graphs could increase the management costs reasonably. In spite of those, KVL is appeared to be well fit to graph traversal algorithms. With link chasing, applications could avoid a lot of overhead during visiting vertices.

4.2 Dynamic web content creation

The rapid increment of the amount of data and the need of improving user experiments make the number of dynamic web pages increase at a high pace. One well-known solution for efficiently delivering dynamic content is to decompose the pages into small fragments and cache those in main memory. Additionally, an object dependence graph (ODG) is constructed to keep track changes and maintain consistency [5, 19]. When a new web page is requested, the ODG is checked to reload fragments whose content has been changed and directly fetch those whose content remains unchanged from the cache. During this process, fragments are issued sequentially since the latter fragment depends on the earlier one. With such kind of access pattern, using the key-value model implemented in popular in-memory caches to store the fragments and ODG could lead to high cumulative overhead when creating a page.

The KVL model is well fit for caching such kind of dataset since it is also a graph in nature. By representing items as fragments and use links to formulate the dependency between fragments, applications could construct the web page by simply chasing links between pages' components. Since link chasing is much faster than item lookup, applying this model would reduce the cumulative latency significantly.

4.3 Intelligent transportation systems

Intelligent Transportation Systems (ITS) act important roles in solving critical issues in urban areas such as congestion, air pollution, and safety of transit. The major problem of ITS system is that they have to manage a huge amount of data which is pushed to the system continuously from many sources like GPS, video stream, etc., in order to produce meaningful information in real-time. To do so, the digital map must be well organized since most of the critical operations such as map matching, routing, and congestion detection relies on it. As the map could be considered as a network of points (e.g. intersections) and lines (e.g. streets), KVL model is a promising candidate for representing its content in ITS systems.

5 KELI: A KVL In-memory Store

We have implemented an in-memory store utilizing the KVL model named KELI (stands for KEy-value-with-Link In-memory Store). Originally, we developed KELI while constructing a traffic condition monitoring system for Ho Chi Minh City (available at *traffic.hcmut.edu.vn*). The main role of KELI is to manage the metadata of the city map so that applications could quickly process GPS signals generated by vehicles to produce meaningful information about the current traffic condition of the city [14]. Although KELI is originally designed for an ITS system, we feel its architecture is general enough for working with other applications. So we extended its implementation to make it applicable to a wide range of applications.

5.1 System architecture

Our objective when designing KELI is to provide a lightweight in-memory store for hardly-changed datasets stored in complicated (disk-based) databases. Particularly, KELI copies items stored in the database to memory and lets applications to access them through its interface instead of sending requests directly to the database. The design of KELI also assumes that update occurs very rarely and a few changes do not cause serious impact on application performance and correctness.

Figure 2 illustrates the overall architecture of KELI. Data is originally stored permanently on disk to ensure durability and availability. KELI is deployed entirely in memory. After starting up, KELI accesses data on disk and loads them into memory. During this process, items are transformed from their original format on disk to KVL format. After KELI finishes loading data from disk, data access could be redirected to KELI and the database now acts as a backup module.

KELI does not support update operations (i.e. modify, write, and delete) so if application want to change the content of data, it still has to send those requests to the database. Updates occurring at disk do not take effect immediately to the in-memory store. KELI, however, reloads the content of data on disk after predefined and fixed intervals.

5.2 Data layout

Since DRAM capacity is much smaller than that of secondary storage, utilizing memory space efficiently is a crucial requirement. To do so, avoiding/reducing fragmentation is necessary as this is the primary source of low memory utilization. S. M. Rumble et al. [20] showed that current standard dynamic memory allocators such as *"malloc"*

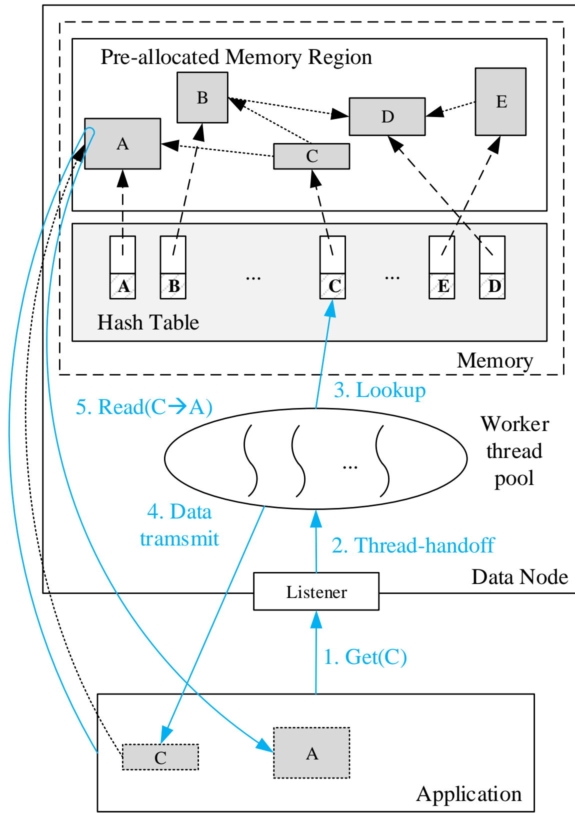


Figure 4: Request Processing

in C do not handle this problem well. Therefore, to avoid fragmentation, we do not utilize dynamic memory allocators to create rooms for data. Free space is instead reserved in advance in form of contiguous memory slots. KELI dramatically fills them up with the content of new items.

KELI updates its content periodically in batch-style. Every time the update process is triggered, KELI first allocates new memory regions for new items then fills them up with the content of data stored in the disk-based database. After that, it deallocates the memory regions of old data and uses items in the new memory regions for answering upcoming requests from applications. As the clients bypasses the server when chasing links, KELI must ensure applications do not access old items after the update took place by halting all active connections from the clients and have the clients reestablish those connections to the server to obtain the new content.

Similar to key-value stores, KELI employs a hash table for tracking items by their key. We use the Cuckoo hashing [18] to implement the hash table since this technique ensures constant complexity in the worst case, guarantees stable performance with large datasets. The hash table does not hold the content of hashed items but it instead stores the pointer to the actual data. So for each new item, KELI first finds a slot in allocated memory regions for it and write its content to this slot. After that, item's key and the pointer to the slot are added to the hash table.

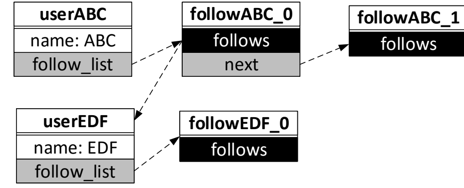
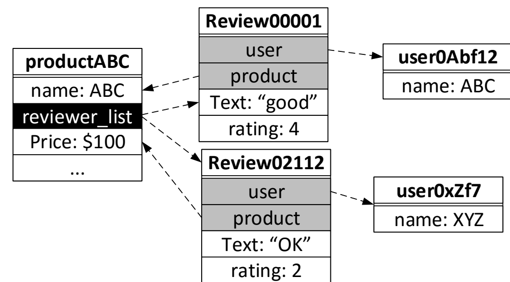


Figure 5: Modeling Twitter datasets by the KVL model. Black boxes represent a list of links and each link refers to one item in the dataset. Gray boxes represent a single link.

Some stores such as HBase [1] keep items in memory in form of memory objects to simplify data management. This approach, however, often requires the server and client to serialize/deserialize those objects from/to an array of bytes before transferring them over the network. Serialization adds significant overhead to request processing especially in small ones like item lookups. Furthermore, if items contain pointers to different resources, chasing links would generate multiple RDMA Reads making the operator inefficient. Therefore, KELI servers store items in form of arrays of bytes and let the client perform serialization.

5.3 Request processing



(a) Fragments represented by KVL model.

(b) Web Page content.

Figure 6: Modeling a dynamic web page by the KVL model. Black boxes represent a list of links and each link refers to one item in the dataset. Gray boxes represent a single link.

In this subsection, we will show how KELI handle requests from clients. The whole process is shown in Figure 4. KELI communication modules are built upon IB verb programming model. Given a key, the client asks for its value by issuing a *"get"* request via *"ib_send"*. The request is received at the server side by a listener which has responsibility for receiving any incoming requests. In order to maximize KELI performance, the listener continuously asks input queue for new requests instead of passively waiting for the queue to inform it about the new message like traditional techniques. Although this approach wastes a lot of CPU cycle for polling input queue, it makes KELI respond to the new request very quickly.

When the listener discovers a new request in the input queue, it then pops the request out and forwards it to a worker thread in the thread pool. Threads are chosen randomly to ensure load-balancing. After receiving a request from the listener, chosen thread then searches for the needed item in the hash table. If the item is not found, it generates a response with empty payload and sends it back to the client using *"ib_recv"* operation. Otherwise, the hash table would return a pointer to the location of the item. Thread just simply follows the pointer, generates a non-empty response message, copies the content of the item to the payload of this message and sends the response back to the client (also using *"ib_recv"*).

The client has responsibility for interpreting the meaning of the payload of the response message. If the item contains links to other items and application wants to retrieve them, the client does not make another *"get"* request but using RDMA Read to directly read the content of the linked items from the server. Doing so significantly reduces item loading latency since executing RDMA Reads is much cheaper than explicitly invoking an item lookup request (e.g. *"get"*).

6 Experiments

6.1 Experiment setup

In this section, we will illustrate the benefits of employing the KVL model for RDMA-based in-memory stores by comparing the performance of KELI with another in-memory key-value store. We choose Memcached for this task due to its popularity. In fact, to make the comparison fair, instead of using the original version, we make use of an extended version of Memcached, which uses RDMA verbs for data transmission, for all experiments. [11, 10]

The two stores are compared based on practical applications. Particularly, we use the KVL model to represent several real-life datasets and let KELI manage them. We do the same task with Memcached except that links in items are replaced by the key of referred items. We then develop some applications implementing popular algorithms working over those datasets. The data such applications need for computation is fetched from either KELI or Memcached. We measure the computation cost and use it to compare the two stores.

6.2 Data modeling

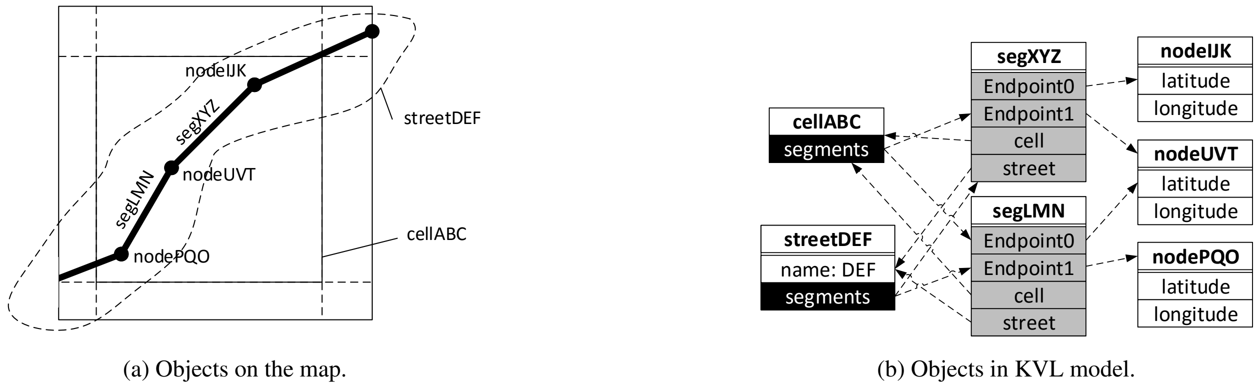
We conduct experiments on three different real-life datasets, each associated with one problem listed in Section 4. In the text bellow, we will illustrate those datasets and describe how to use the KVL model to model them. For the key-value version, we just replace links by the key of item it pointing to.

Social Network Graph traversal is very popular on the social network. For examples, given a user, find a person with a given name (e.g. "John") among his friends, his friends' friends, and so on is a typical problem making use of graph exploration. In the experiment, we will perform the Breadth-First-Search (BFS) over a real-life social network dataset provided by Twitter. The dataset contains about one million nodes represent users and more than 22 million edges represent the followership between users. Figure 5 shows how the dataset is modeled by the KVL model. Clearly, this representation is similar to adjacent list data structure except for that edge (e.g. follower) lists are broken into multiple chunks since one user may have a lot of friends. If we integrate all of them into one item, this could enlarge the size of this item reasonable leading to performance degradation. In following experiments, we let each list contain at most 100 followers.

Web Page Generation We construct a web page displaying information about the reviews of products sold by Amazon using the dataset provided by Amazon itself. The content of the page is dynamic as product information change frequently and users continuously update their reviews to products. We have to break the HTML file into multiple parts and change their content right after the update takes effect. Figure 6a shows the relationship between users, products, and reviews of users for some products and Figure 6a shows how the HTML file of the page lock and feel.

Map Matching We choose map matching problems as a representative application for ITS systems. Given a GPS signal, we have to determine if this signal belongs to any street and if so, identify which place on the street it falling into. This problem is very popular in ITS system involving to real-time traffic monitoring, congesting detection, routing, etc.

In this experiment, we use a digital map provided by OpenStreetMap (OSM) to construct the datasets about streets in Ho Chi Minh City. Figure 7 illustrates an example of modeling the map by the KVL model. Particularly, according to OSM's format, a street is a polyline which is constructed by connecting multiple nodes (points). Since the street is a polyline and typically long, we do not map GPS signals with streets but with lines which are constructed by connecting two consecutive nodes on a street called segment. An item represents a segment will link to items containing the information about its endpoints



(a) Objects on the map. (b) Objects in KVL model.

Figure 7: Modeling objects on digital map by the KVL model. Black boxes represent a list of links and each link refers to one item in the dataset. Gray boxes represent a single link.

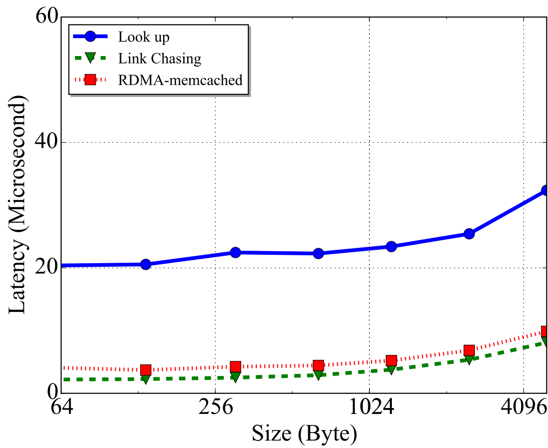


Figure 8: The latency of link chasing and item lookup in experiments.

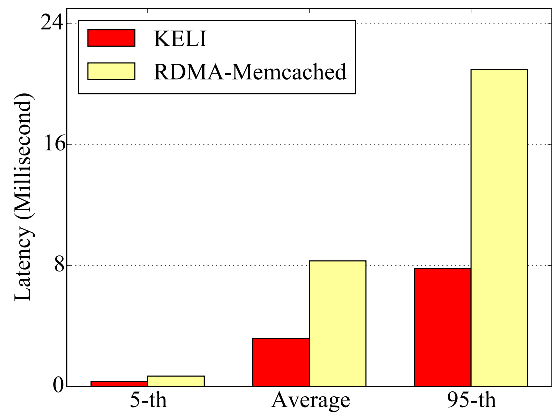


Figure 9: Map matching latency

(node). There are also links from streets to segments constructed from their nodes. We group segments into disjoint areas called cells based on their geographical location. The map matching algorithm is quite simple: given a GPS signal, the application first determines its spatial information (e.g. latitude and longitude) and uses them to identify the corresponding cell. It then issues the in-memory stores for this cell and then retrieves segments belonging to this cell to find out which segment this signal belongs based on their geographical locations.

6.3 Performance evaluation

We conduct all experiment on two computers equipped with Intel Xeon E5-2670 and 32GB main memory. They are connected through an Infiniband connection using Mellanox's ConnectX-3 40 Gbps NIC. The RDMA-Memcached in all experiments is based on Memcached version 1.4.24 and applications use libMemcached version 1.0.18 to communicate with the store. In order to fully understand the effect of using KVL model, let us first compare the read performance of KELI's item lookup and link chasing with RDMA-Memcached's get operation. Figure 8 shows the experiment results. Clearly, the naive implementation of lookup using Send/Recv verbs performs very poorly. It takes about three to four times slower than the optimized version used by RDMA-Memcached. However, item lookup still executes two times longer than link chasing. Therefore, if applications make good use of link , KELI could perform better than RDMA-Memcached.

Although KELI has to deserialize item content and check for consistency when chasing links, link chasing latency is just slightly slower to that of pure RDMA Read reported in Figure 3. This is because the time spent on communication is the dominant cost of RDMA operators. So although KELI has to check for consistency and deserialize every item it reads, its latency is still lower than that of HERD. Also note that HERD's lockup latency could be higher in practical as it sacrifices reliability and let applications take care of integrity checks to boost the lookup performance as much as possible.

In the map matching experiment, we preload both KELI and RDMA-Memcached with about six million key-value

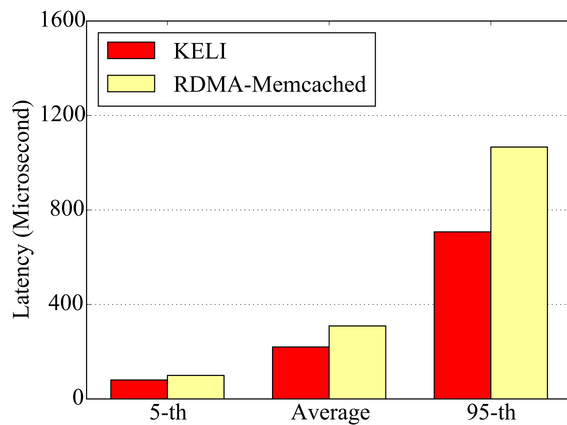


Figure 10: Web page construction latency

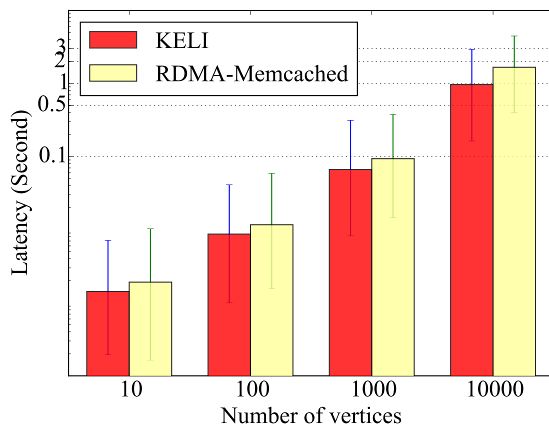


Figure 11: Graph exploration

pairs represent the geographical information of Ho Chi Minh City. Similarly, we prepare about 200 thousand reviews for more than 12 thousand products for web creation application and a graph with one million nodes and about 22 million edges for BFS traversal. Figure 9, 10, and 11 show the execution time of map matching, web page construction, and BFS algorithms, respectively, using KELI and RDMA-Memcached.

Apparently, KELI outperforms RDMA-Memcached in all cases. In the case of map matching, KELI outperforms RDMA-Memcached by the factor of two in average. In the case of tail latency (95-th percentile), KELI still runs about 2.5 times faster than RDMA-Memcache. KELI also helps applications construct web pages 50% faster than RDMA does. Similarly, the implementation of BFS algorithm using KELI runs 75% faster than that using RDMA-Memcached.

The reason behind this is that according to the data layouts we described in the previous section, applications utilizing KELI mostly uses link chasing for fetching new items. For example, in the case of graph traversal, the application only has to invoke item lookup for the first time when it has to retrieve the first vertex. After that, based on the "list" and "next" links integrated into each accessed vertex and edge lists, the applications could always invoke link chasing to get information about vertex to be accessed. On the other hand, applications supported by RDMA-Memcached have no choice but item lookup to retrieve data. Since this operation is about two times lower than link chasing, KELI performs two times better than RDMA-Memcached.

7 Conclusion

In this paper, we present KVL, an enhanced version of the key-value model for in-memory stores working over RDMA-capable networks. In this model, each data set is a network of key-value pairs linking to each other. Each link is a pointer to the address of the referred item and is integrated directly into the item. With this organization, the KVL model introduces a new operation named link chasing to allow applications to utilize RDMA Read to directly read items through links without interfering the data server. Our experiments have shown that this model is well fit many real-life applications. Also, by utilizing this model, KELI, an average in-memory store without much optimization could easily outperform an state-of-the-art in-memory store.

.

References

[1] Hbase. https://hbase.apache.org/. Accessed: 2017-03-03.

[2] Memcached. https://memcached.org/. Accessed: 2016-11-07.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[5] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 844–853. IEEE, 2000.

[6] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 26. ACM, 2016.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[9] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[10] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K Panda. Scalable Memcached Design for Infiniband Clusters Using Hybrid Transports. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 236–243. IEEE, 2012.

[11] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. Memcached Design on High Performance RDMA Capable Interconnects. In *2011 International Conference on Parallel Processing*, pages 743–752. IEEE, 2011.

[12] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA Efficiently for Key-Value Services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[13] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[14] Minh Duc Le, The De Vu, Duc Hieu Nguyen, Tien Hai Ho, Duc Hai Nguyen, Tran Vu Pham, et al. Keli: a key-value-with-links in-memory store for realtime applications. In *Proceedings of the Seventh Symposium on Information and Communication Technology*, pages 195–201. ACM, 2016.

[15] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[16] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, 2010.

[17] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[18] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[19] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic Detection of Fragments in Dynamically Generated Web Pages. In *Proceedings of the 13th international conference on World Wide Web*, pages 443–454. ACM, 2004.

[20] Stephen M Rumble, Ankita Kejriwal, and John K Ousterhout. Log-structured memory for dram-based storage. In *FAST*, volume 14, pages 1–16, 2014.

[21] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.

[22] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. HydraDB: A Resilient RDMA-driven Key-Value Middleware for in-Memory Cluster Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 22. ACM, 2015.

[23] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.