Multi-threading in-memory key-value storage

Candidate Number: 8
Energy Informatics - Green Computing
INF-3210-1 21H

Department of Computer Science
UiT, The Arctic University of Norway

Autumn 2021

CONTENTS

ABSTRACT

In-memory key-value store has received a lot of attention in recent years due to its ease of understanding, better throughput, and low-latency access compared to other stores. Although these types of systems are more efficient, but optimal memory consumption, improved processing speed, parallelization of processes and also reduce power consumption are the existence challenges in designing such systems. In this report, we design and develop multi-threading in-memory key-value store system. It uses Trie-tree as the main data structure. It adopts spin-lock mechanism to avoid race conditions in the system and guarantee mutual exclusion. Experimental results show that the proposed system has an acceptable and defensible performance in the field of energy efficiency under multi-threading mode.

1 INTRODUCTION

hKey-value stores (KVS) are a vital component of many systems tat need high throughput and low-latency access to large volumes of data [1]. Among these systems and applications, we can mention web indexing [2], search engine [3], social networking [4, 5], photo stores [6], e-commerce [7], on-line gaming [8], and etc. Indeed, a key-value store is a simple database that uses some data structure to store keys and values such that each key is associated with one and only one value in a collection. This relationship is referred to as a key-value pair. In each key-value pair the key is represented by an arbitrary string such as a hash, URI, or filename. The value can be any type of data like an picture, document, or user preference file [9]. Figure 1 illustrates a simple schematic view of the structure of Key-value stores. Where each Key is associated with only one value.

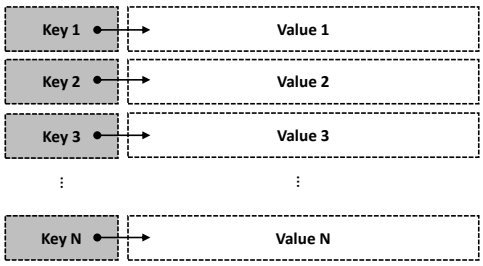


Figure 1: Schematic view of the structure of Key-value stores

In general, key-value store (KVS) systems divided into two categories: in-memory key-value store and disk-based key-value store. In in-memory key-value store, the data will be stored in the memory cache and they are not permanent. It means that the data will be lost when the power is cut off or disrupted. In contrast, in a disk-based key-value store, data is stored on the hard disk persistently. But in terms of performance, in-memory key-value stores have better and faster performance than disk-based key-value stores due to faster memory speed than disk. Therefore, this report intends to describe the design and implementation of an multi-threading in-memory key-value store from scratch and also evaluate it from the energy efficiency aspect. The proposed in-memory KVS uses the Trie-tree [10] as the main in-memory data structure for achieving high insert and lookup performance. For concurrency, the proposed KVS uses Spin-lock mechanism to avoid race conditions and guarantee mutual exclusion. Also for benchmarking and evaluating the proposed KVS, two libraries have been used. For energy-efficiency evaluation, POET [11] have been integrated to the proposed KVS and Yahoo! Cloud Serving Benchmark (YCSB) [12] have been used for benchmarking. So, the rest of this report is organized as follows. Section 2 briefly discuss background of data structures that used for KVS. Section 3 presents the design and implementation of KVS. Section 4 describes the experimental methodology. Section 5 will discusses experimental results. Finally, the report concludes in Section 6.

2 BACKGROUND

There are several data structures to design a KVS system. Some of them are very efficient and some of them are not. In the following, some of these data structures will be examined.

B-Trees

B-tree is a balanced multi-way search tree data structure that maintains sorted data. The B-tree is an extended binary search tree data structure that allows nodes with more than two children [13]. The B-tree is more used for storage systems which needs to read and write relatively large blocks of data. Therefore, B trees are not very efficient, when they reside in RAM. Inserting or deleting node involve moving many keys/values around. if a B-tree has a very low branching factor (the number of intermediary nodes), then it has a chance to be efficient in RAM. This may lead to cache miss minimization. Leaf and non-leaf nodes are of different size, this complicates storage. The

time complexity of the B-tree is logarithmic in all operations. This means that B-trees have $O(\log n)$ complexity where n is the number of nodes in the tree. Another type of B-Trees that can be used for KVS systems is is B^+-Tree. In fact, is B^+-Tree is an extension of B-Tree. The difference between them is that B^+-tree stores the records as leaf nodes and stores the keys only in internal nodes. While, B-trees stores keys and records as internal as well as leaf nodes [13]. The records are organized into a linked list fashion which allow B^+-tree makes the searches more faster and efficient. The disadvantages of this data structure is related to the extra insertion and deletion overhead, space overhead. In addition to the previous two B-Trees, there is also a B^*-tree. B^*-tree is a special form of B^+-tree. The difference is that, the number of points in each node in B^+-tree is half $(1/2)$ of it's maximum capacity. While in B^*-tree this number is at least $2/3$ full [13]. This makes the B^*-tree more compact than the B^+-tree.

AVL Tree

AVL tree is a height balanced binary search tree where the difference between heights of right and left subtrees cannot be more than one for all nodes [14, 15]. In the AVL tree, each node has a Balance Factor (BF) that is calculated by subtracting the height of its right subtree from the height below its left tree [15]. So this BF will be the numbers of -1, 0, 1. If BF is 1, it means that the left sub-tree is one unit higher than right sub-trees. If BF is equal to 0, it means that both left and right sub-trees have the same height and the tree is balanced. If BF is -1, it means that the right sub-tree is one unit higher than left sub-trees. Otherwise, if the BF is not in the range of -1 and 1, the tree will be unbalanced and need to be balanced. The time complexity of this data structure, for all operations (Insert, Delete, and Lookup), is equal to $O(\log n)$. Where n is the number of nodes. The space complexity of AVL tree is equal to $O(n)$. However, since this data structure always tries to keep the balance in the insert and delete operations, so in the lookup operation, this data structure can be efficient. But on the other hand, due to the fact that the rotations in this data structure is very high, this increases the overhead pointer which leads to an increase in space. Also, from an implementation point of view, this tree has a more complex implementation than other trees.

Hash Table

Another data structure that is widely used in various systems is the hash table. A hash table is a type of data structure that associates the values with special keys with a hash function [16]. The basic operation that this table facilitates is the lookup operation. This means that the user can quickly find the desired data in it. In hash tables it is also possible to add new data in a short time. The time required to lookup and add depends on the type of table and the amount of data. This time can reach the time order $O(1)$ by selecting the appropriate table. But on the other hand, this data structure also has disadvantages. In most hash functions, it is assumed that a collision occurs naturally, that is, two different keys find the same amount of hash and thus enter a cell of the array [16]. Therefore, hash tables become quite inefficient when there are many collisions. In addition, operations on a hash table take constant time on average. Thus hash tables can not be effective when the number of entries is very small. It is slow due to synchronization.

Given the above description of the various data structures, the data structure used for this proposed system is the Trie-tree data structure [10]. This data structure, in addition to being very robust, has a much easier implementation than the other structures mentioned. A description of the trie tree will be provided in section 3.2 in detail.

3 DESIGN AND ARCHITECTURE

3.1 Architecture

Proposed KVS is a user space key-value store system based on memory architecture. It supports the ordinary KVS system operations and interfaces, such as **PUT** and **GET**. Figure 2 illustrates the architecture of the proposed KVS system. Based on this architecture, the application or user sends its request to the proposed KVS system as an interface. The proposed system also processes

this request and returns the result to the application or user. These put and get operations are designed to be very simple in the terms of signature, so that the user only needs to enter the key and the corresponding value for insertion and only needs a key to lookup. In the following and in subsection 3.3, this operation will be described in detail.

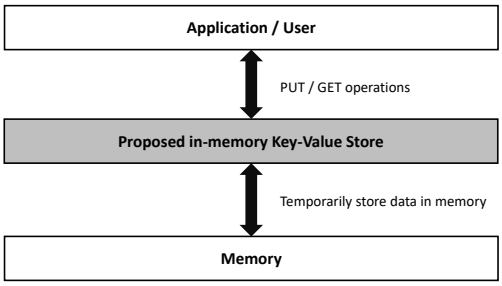


Figure 2: The architecture of the proposed Key-value store

3.2 Data Structure

The data structure that selected for design the in-memory key-value store is a **Trie tree** [10]. A trie is a multi-way tree like structure that store sets of strings by successively partitioning them. Trie tree have several appealing properties: 1) Deals well with strings and uses less memory. 2) Search for your key depends on the length of the key and not the number of nodes in the trie. 3) Height of a tree is independent of the number of keys it contains. 4) Requires no re-balancing when updated and 5) The Complexity of all functionalities are $O(h)$ where h is the height of the tree. These characteristics make the trie tree a better option for design the in-memory key value store. In the trie structure the key is always a string, but the value could be of any type, as the trie just stores. Figure 3 shows naïve trie tree that illustrate how this data structure keep key as prefix characters.

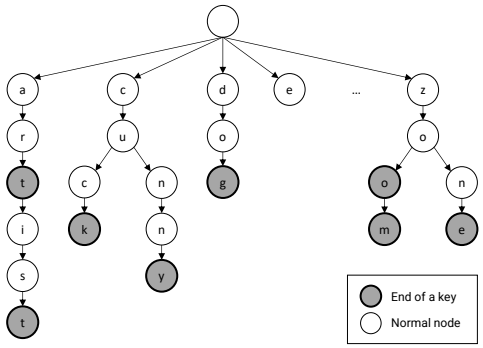


Figure 3: A simple and naïve view of the trie tree structure

The gray nodes represent if the current node is the end of a string (key). For example, in Figure 3, the '**art**' as a key is divided into separate characters, and the character '**t**' is shown as the end node of this key in gray color. Also, if there is another key that the word '**art**' is part of it, that key will continue from the end of the previous key (i.e, art). In this way, searching for keys in this structure will be much easier and more effective. In addition, each node in the trie has 52 children. These children are in the form of alphabetic characters. Each path from the root to the node represents a string i.e; a key and the node stores the corresponding value of the key. Furthermore, the proposed key-value store include some constraints that have been observed and considered in the design. The first constraint is related to the key size, which in this system is equal to a maximum of 64 bytes. Because the time complexity of the trie data structure is directly related to the length of the string (key), thus, the 64 bytes key length is a suitable trade-off for this system. The second constraint that is applied is that the characters within a string (key) are between [a-z] and [A-Z]. This means that the key stored in this data structure can only be a string of letters of the alphabet. A string that does

not contain numbers as well as special characters. The third and last constraint is the value size, which considered as 256 bytes.The larger size of the value, leads to the more memory occupied, and since this KVS system is designed for testing, 256 bytes is suitable for storing the corresponding values to the keys. The data that can be stored in each trie node is described in Appendix A.

3.3 Functionality

As mentioned earlier, in architecture subsection 3.1, the proposed KVS system includes two simple interfaces (PUT and GET), to give the user and the application the ability to easily interact with the system. In the following, these interfaces will be described.

3.3.1 Put operation

The first function is related to the insertion (PUT) operation. For this purpose, traverses through the trie to insert the value to the corresponding key. At this point there are five possible cases,

- **Case 1:** Key string to be putted doesn't exist and had no common prefix in the data structure.
  In this case the key string is simply putted via a pointer from the root node.
- **Case 2:** Key string to be putted already exits.
  Here the trie-tree traversed to find the last node of the key string and the value is overwritten.
- **Case 3:** The key string to be putted is a super string of a node present in the trie-tree.
  Here, the common prefix is remains untouched and the rest part is appended to this common prefix.
- **Case 4:** The key string to be putted is a sub-string of a node in the trie-tree.
  Here, the existing part is broken into the common prefix and the remaining part which is append to the common prefix node.
- **Case 5:** The key to be putted has a common prefix with an existing nodes in the trie-tree.
  Here, the uncommon part of the key string is broken into a new node and the key is appended to the common prefix.

The complexity of this functionality is $O(h)$ where h is the height of the tree. Since in the worst case, we have to traverse from the root to the leaf.

3.3.2 Get operation

To find a key and its corresponding value, traverses through the trie tree structure to find the key. If the key is found, then the value corresponding to that key will be returned as a result. If the key is not found, the empty value will be returned. The complexity of this functionality is $O(h)$ where h is the height of the tree [10]. Because in the worst case, we have to traverse from the root to the leaf.

3.4 Concurrency Control

Proposed KVS system supports multi-thread mode. This system adopts the spin-lock mechanism [17, 18] as a main approach to avoid race conditions and guarantee mutual exclusion in the multi-threading scenario. In general, trie tree structures do not need to apply mutual exclusion mechanisms to lookup a specific key. This data structure only needs mutual exclusion control during insertion operation. This is because in this operation, the race condition between threads to create a node at the same time could be happen. For this purpose, instead of global lock strategies (e.g., mutex) which has very bad affect on performance of system, fine-grained lock was used. Fine-grained lock means that the lock applies to only a small part of the trie tree structure and not the whole tree. If a new node needs to be created and added, its parent node will be locked until a new node is created and added. This simple way prevents the other threads ,that are working on the other nodes, from being involved in the locking mechanism. Therefore, the **Test-and-Test-and-Set** mechanism [18] has been used to guarantee mutual exclusion in proposed system. The code of this mechanism is described in Appendix B.

4 IMPLEMENTATION

The core of proposed system is written and implemented in C programming language. The C++ language has also been used to evaluate and benchmark the system. POET [19] library has been used for optimize energy consumption of this system. To use POET, it must be integrated into the benchmark program. For integration, in addition to the need for a number of functions to be called within the proposed system, there are two configuration files (CPUs configuration and control configuration) need to be modified. The CPU configuration file contains information about the CPU frequency as well as the number of desired processors. Therefore, frequencies between 1.2 GHz and 2.0 GHz are set. All these frequencies are intended for the number of cores 1 to 4. After that, the SppedUp factor and PowerUp factor are calculated and located in another configuration file, called control configuration. One of the important reasons we used multiple frequencies is that POET can perform optimization with higher accuracy. But on the other hand, it can increase the computational overhead by POET.

5 EVALUATION

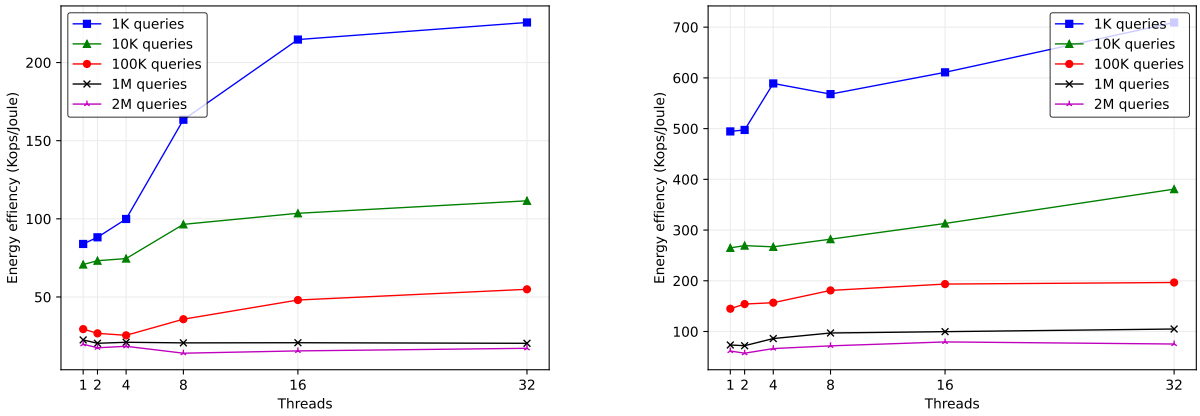
5.1 Experimental Setup

POET Toolkit perform based on processors that support Dynamic Voltage/Frequency scaling (DVFS) technology. Therefore, all experiments are conducted on the Linux machine (server) supplied by UiT. The server has an **Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50GHz** CPU with 6 cores and all cores support DVFS technology and can be adjusted between 1.2 - 2.5 GHz. The main memory size is 64 GB. The Operating System running on this server is Ubuntu 20.04.3 LTS (Focal). In addition, for benchmarking and evaluation, the YCSB [1] [12] used as the benchmark tool.

5.2 Evaluation Result

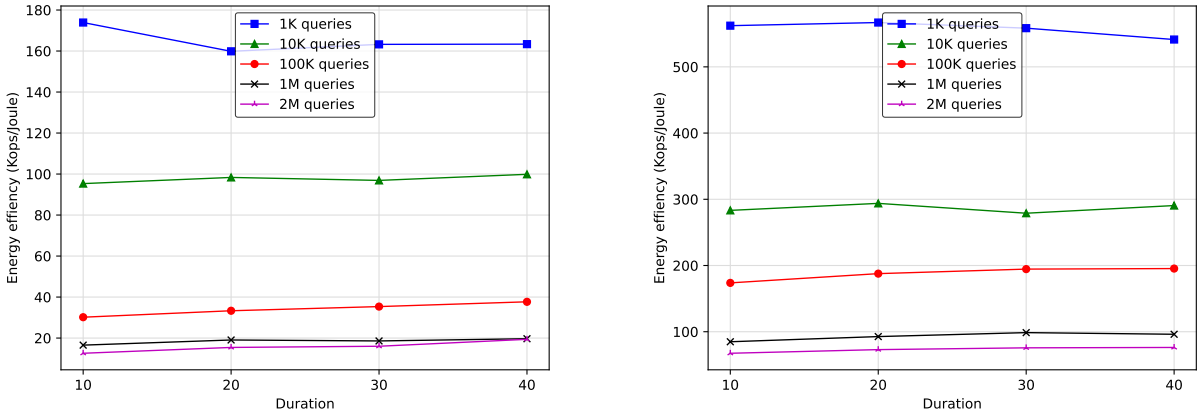
In this section, the proposed system will be evaluated from the perspective of energy efficiency. For this purpose, we use workloads A and B of the YCSB benchmark to produce two types of read/update ratio: 50:50 percent and 95:5 percent (heavy read). In addition the type of Zipfian used as the access pattern. We also generated five data types for each YCSB benchmark Which ranges from a thousand to two million queries. Indeed, evaluation, in terms of energy efficiency, is examined in two ways. The first is when the system is run in multiple threads for a specified duration, and the second one is when the system is run for a specified number of threads at different duration. For the first way, we scale the working threads from 1 to 32 and run the proposed system for duration 30 seconds and for five data types to observe the energy efficiency change under multi-threading mode. The results of this evaluation are shown in Figure 4. Indeed, energy efficiency is defined as the number of operations on the amount of energy consumption (in joules). As can be seen in Figure 4, in both 50:50 and 95:5 percent data type, the energy efficiency of the system increased or remained relatively constant when the working threads increase from 1 to 32. The reason for this is related to the use of the Trie-tree data structure, which it has a certain time complexity in all operations and is independent of the number of nodes in the system. Therefore, as the number of nodes increases, the performance of the system does not decrease significantly and in worst case remains constant. Also, by looking at the diagrams, we can see that the energy efficiency of the system at 95:5% data type, in Figure 4b, is higher than 50:50%, in Figure 4a. The reason for this is related to the fact that in a read-heavy workload (95:5%), the proposed system easy to handle the same IO request by multi threads. As can be seen in the diagrams, as the number of queries increases, the energy efficiency of the system decreases. Important reasons for this include configurations for the POET toolkit. Since we set different frequencies for different number of cores and also speedup and powerup factors, it can cause a computational overhead in the POET. Because the POET uses mathematical optimization and linear programming, so as the number of constraints increases, the optimization time also increases, and this can affect the overall number of operations in the system and further affect energy efficiency.

1 The version used is version 0.17.0



(a) Zipfian GET 50% **(b)** Zipfian GET 95%

Figure 4: Energy efficiency on different number of threads for 30 seconds of duration



(a) Zipfian GET 50% **(b)** Zipfian GET 95%

Figure 5: Energy efficiency on different duration for 8 threads

The second part of the evaluation is related to the review of energy efficiency for different running duration. For this purpose, we run the system for different duration (from 10 to 40 seconds) and for fixed 8-thread mode. The results of this benchmarking can be seen in Figure 5. As can be seen, with increasing duration, the energy efficiency of the system remains almost constant. One of the reasons is related to the Trie-tree data structure. Because this data structure is robust and does not need re-balance in general. Another reason could be related to the use of the POET toolkit. Where it selects the most optimal available frequency for a number of different cores. In this way, by minimizing energy consumption, it directly affects energy efficiency of the system.

6 CONCLUSION

In this report, we present a robus in-memory key value storage system that uses the Trie-tree data structure as an efficient data structure for this purpose. The proposed system adopts a spin-lock mechanism to avoid race conditions and guarantee the mutual exclusion. The system was evaluated in terms of energy consumption. For bencgmarking and evaluation, the YCSB version 0.17.0 and POET toolkit have been used. Experimental results showed that the proposed system has acceptable energy efficiency in the multi-threading mode. The energy efficiency of this system has remained almost constant or increasing with increasing number of threads and running duration.

REFERENCES

[1] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.

[2] Xiang Xu, Fumin Zou, Lvchao Liao, and Quan-Ling Zhu. Experimental analysis for calculation performance of mass data based on gemfire: Experimental analysis for calculation performance of mass data based on gemfire. *Journal of Computer Applications*, 33:226–229, 2013.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), jun 2008.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 47–60, USA, 2010. USENIX Association.

[5] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 18, USA, 2012. USENIX Association.

[6] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015.

[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. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1), mar 2017.

[9] what is a key value store. https://aerospike.com/what-is-a-key-value-store/.

[10] Data Structures, Algorithms, and Applications in Java. https://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/tries.htm.

[11] Connor Imes, David H.K. Kim, Martina Maggio, and Henry Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86, 2015.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.

[14] Caxton C. Foster. A generalization of avl trees. *Commun. ACM*, 16(8):513–517, aug 1973.

[15] D. Šuníková, Z. Kubincová, and M. Byrtus. A mobile game to teach avl trees. In *2018 16th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 541–544, 2018.

[16] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery.

[17] T.E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[18] B.N. Bershad. Practical considerations for non-blocking concurrent objects. In *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*, pages 264–273, 1993.

[19] The Performance with Optimal Energy Toolkit. http://people.cs.uchicago.edu/~ckimes/poet/.

APPENDIX A

In the code below, the information stored in each node of trie-tree is displayed.

```
#define ALPHABET_SIZE (56)

struct TrieNode
{
  struct TrieNode *children[ALPHABET_SIZE];
  bool isEndOfWord;
  char* value;
  bool lockHeld;
};
```

Where the **children** Contains the pointers to the children. Indeed, each node represents a key obtained by the concatenation of all the characters of it's Ancestors. **isEndOfWord** is a variable that indicates whether this node is the end node of the key. Variable **value** also contains the value corresponding to the key. **lockHeld** variable is used for handle concurrency in the system. If this value is true, it means that a thread has already entered the Critical section and the other threads can not continue with this node and will have to wait.

APENDIX B

Below you can see part of the code related to locking mechanism which used in proposed system for handle mutual exclusion.

```
static void acquire_lock(struct TrieNode *node) {
  while(true) {
     while(node->lockHeld == true);
     if(!test_and_set(&node->lockHeld)) break;
   }
   return;
}

static void release_lock(struct TrieNode *node) {
  node->lockHeld = false;
  return;
}

static bool test_and_set(bool *flag) {
  bool prior = *flag;
  *flag = true;
  return prior;
}
```

In the above mechanism, the lock is finely applied to the nodes of the trie tree and not the whole tree. This way, in addition to ensuring mutual exclusion, there will be no disruption to the KVS engine performance. There is also a similar mechanism called **test-and-set**, which is a kind of spin-lock mechanism. But the **test and test-and-set** mechanism performs better in terms of overhead than the **test-and-set** mechanism. Hence, the **test and test-and-set** mechanism has been chosen for this proposed KVS system.