Article

Research on Multicore Key-Value Storage System for Domain Name Storage

Luchao Han ^{1,2} , Zhichuan Guo ^{1,2,*} and Xuewen Zeng ^{1,2}

¹ The Institute of Acoustics of the Chinese Academy of Sciences, 1921 North Fourth Ring West Road, Haidian District, Beijing 100190, China; hanlc@dsp.ac.cn (L.H.); zengxw@dsp.ac.cn (X.Z.)

² School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100049, China

* Correspondence: guozc@dsp.ac.cn; Tel.: +86-1861-206-2648

Abstract: This article proposes a domain name caching method for the multicore network-traffic capture system, which significantly improves insert latency, throughput and hit rate. The caching method is composed of caching replacement algorithm, cache set method. The method is easy to implement, low in deployment cost, and suitable for various multicore caching systems. Moreover, it can reduce the use of locks by changing data structures and algorithms. Experimental results show that compared with other caching system, our proposed method reaches the highest throughput under multiple cores, which indicates that the cache method we proposed is best suited for domain name caching.

Keywords: multicore; key-value store; cache replacement algorithms

Citation: Han, L.; Guo, Z.; Zeng, X. Research on Multicore Key-Value Storage System for Domain Name Storage. *Appl. Sci.* **2021**, *11*, 7425. <https://doi.org/10.3390/app11167425>

Academic Editor: Paolino Di Felice

Received: 16 July 2021

Accepted: 4 August 2021

Published: 12 August 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Domain Name System (DNS) can direct web traffic to the correct destination. It is used by everyone, everywhere, and all Internet traffic flows through it. In the field of network security, IP is the most important means of distinguishing tracing network attacks. However, it is difficult for network security managers to remember a series of numbers when manually identifying IP, so the corresponding domain name needs to be displayed behind the IP. At this time, it is necessary to add a DNS cache system to the network-traffic capture system to achieve this function. The best way to achieve quick query of IP and domain names is to use a key-value pair caching system.

Key-value storage usually stores data in memory to accelerate hard disk based caching systems that are slow to read and write. Throughput and hit rate are the two most important indexes to evaluate the performance of this key-value storage method. The cache system usually improves the cache-hit rate by improving the data structure and cache replacement strategy. However, it is difficult to improve the performance of key-value storage while ensuring a high hit rate simultaneously. For example, using the FIFO replacement strategy can significantly improve the performance of key-value storage, but results in very low hit rate.

We find that the DNS cache is the main performance bottleneck of the network-traffic capture system. When the network-traffic capture system works, it needs to store the IP and domain name key-value pair in the DNS response packet in the traffic. When analysing network traffic, the IP address in the traffic in the key-value pair cache is queried to obtain the corresponding domain name. After that, the network-traffic capture system stores the IP and domain names in the corresponding log files. In this way, network administrators can intuitively see IP and domain names when querying the network-traffic log files. When the network-traffic capture system analyses the network data packet in real time, it has extremely high requirements on the performance of the cache query, especially the query latency. The overall throughput of our system is 20 Gbps, and the packet rate is

1 Mpps. Since each data packet needs to be queried once, and the time-consuming query operation will increase in each data packet operation, at least 10 Mops query speed is required to reduce the impact on the overall system performance. However, the latency of the traditional and more popular memory cache system, Memcached, cannot meet our requirements. The improved cache system, memc3, does not perform well in the case of multicore. Therefore, it is necessary to develop a low-latency, in-memory key-value pair caching system in this paper.

The main contributions of our work are summarised as follows. 1. We proposed a key-value pair caching architecture for DNS caching, which separates the read and write operations. The write operation is only performed in the main table, which summarizes the information of all cores, and the read operation is only performed in the proxy table, which only contains the information on its own core. This design can significantly reduce the locks' impact on system performance. 2. We reduce the time of synchronization operation by separating each cache table into multiple sub-tables. Each sub-table is a separate linked list in terms of data structure. In this way, when the proxy table is synchronized by the main table, only the changed sub-tables need to be synchronized to reduce the synchronization operation time. 3. We designed a replacement algorithm, which can achieve high-speed and low-latency operations while ensuring that high-hot cache items are not replaced.

2. Related Work

2.1. Memory Cache

Memory caching is widely used in web caching and network security. The most famous of these are Redis and Memcache. In addition, a large number of researchers have proposed many new caching systems for different usage scenarios. B. Atikoglu et al. collected detailed traces from Facebook's Memcached deployment and analysed the workloads from multiple angles [1]. According to their analysis, the GET/SET ratio was 30:1, which is higher than assumed in the literature. Pmemcached is an improvement of Memcached [2]. It not only improves the overall performance of the application's persistence layer but also greatly reduces the "warm-up" time required for the application after a restart.

Kim et al. have proposed a real-time cache management framework for multicore virtualisation, which can allocate real-time caches for each task running in a virtual machine (VM) [3]. This is a new type of cache management method. Based on Spark, a unified analytics engine for big data processing, Ho et al. have proposed an efficient and scalable cache update technology [3]. This technology improves the data processing speed of Spark by optimising the memory cache speed. A. Blankstein et al. have designed a new caching algorithm, hyperbolic caching that optimises current caching systems, improves throughput performance, and reduces miss rates [4]. Most of these memory caches have greatly improved read and write performance, but their cache hit rate is relatively low. Some of them do not even have a cache replacement strategy, and directly cache all items.

2.2. Key-Value Store

Research on key-value caching is important to promote faster internet services. There are many mature key-value storage systems that have been commercialised, such as Dynamo [5]. Masstree [6] is a high-speed key-value pair database, which achieves the purpose of quickly and effectively processing secondary keys of any length by optimising the tree concatenation in the data structure. F. Wu et al. have proposed an adaptive key-value pair storage scheme, namely AC-Key [7]. AC-Key increases the adaptability and performance of the cache system by adjusting the size of the key-value cache, key pointer cache, and block cache. Y. Tokusashi et al. have introduced FPGA-based hardware customisation to improve the performance of key-value pair caching [8]. They solve the problem of DRAM capacity limitation on FPGA by proposing a new multilayer cache architecture. X. Jin et al. have introduced a new key-value pair caching system NetCache,

which uses some of the features of programmable switches to improve the performance of the caching system in response to hot queries [9].

X. Wu et al. have designed and implemented a key-value pair caching system called zExpander, which improves memory efficiency by dynamically partitioning cache regions [10]. Y. Chen et al. have introduced FlatStore, a key-value pair caching system that enables fast caching on a single server node [11]. Flashield is a hybrid key-value caching system that uses machine learning to determine whether cached content should be stored in DRAM or SSD [12]. L. Chen et al. have added security and privacy features to the key-value caching system, providing data isolation for different users [13].

For the key-value pair cache, the cache replacement strategy is very important. The cache replacement strategy has a huge impact on the hit rate and cache throughput. The most common caching strategy is least recently used (LRU) caching, which eliminates the least recently used cache items first. J. Yang et al. have collected a large amount of data in the Twitter cache cluster and used the data to analyse and research the cache [14]. Their research has shown that the cache replacement strategy has a huge impact on the cache effect. Considerable research is based on the LRU cache to make further improvements. Y. Wang et al. [15] have proposed an intelligent cache replacement strategy based on logistic regression algorithm for picture storage and communication systems (PACS). Logistic regression algorithms can be used to predict future access rules, thereby improving cache performance.

For the research of cache replacement strategy, simulation and experimental methods are very important. C. Waldspurger et al. [16] have proposed a dynamic optimisation framework that uses multiple scaled-down simulations to explore candidate cache configurations. Z. Shen et al. have designed a new caching system for flash-based storage devices [17]. By using key-value pair caching application scenarios and flash memory device characteristics, they can maximise ring cache efficiency on flash memory devices and only reduce its shortcomings. Y. Jia et al. have proposed a dynamic online compression scheme SlimCache [18]. SlimCache can dynamically adjust the cache space capacity in real time to improve the hit rate.

All the key-value pair caching systems mentioned above performs well under single core, but present poor performance under conditions of high concurrency with multiple cores. Some of the systems can run in a distributed manner, but their distributed optimization methods are not suitable for a single machine with multiple cores. Therefore, it is necessary to optimize the key-value cache system to make it applicable to multi-core environment.

2.3. Concurrency Control

Regardless of whether it is a distributed architecture or a multicore architecture, the cache system has to face concurrency issues [19]. Y. Xing et al. have done a lot of research and experiments on multicore cache systems, and discussed the problem of cache error sharing overhead [20]. MemC3 uses an optimised cuckoo hash to solve the problem of long lock times in high concurrency states [21]. FASTER also uses optimised hash indexes to improve cache throughput performance [22]. MICA, on the other hand, is optimised for multicore architectures by enabling parallel access to partitioned data [23].

Improving or designing new hash tables for high concurrent caching problems is also an important research direction. CPHash is a hash table specifically designed for multicore concurrency scenarios, which uses finer-grained locks to reduce cache misses [24].

2.4. Multiversion Concurrency

Multiversion control is widely studied in distributed systems. One of the most commonly used methods is replication, which is to copy multiple copies of the same data backup. This method can improve read performance, but it brings consistency issues. The backup algorithm is divided into synchronous replication and asynchronous replication. Among them, the synchronous replication performance is poor, and the loss of any node

in the system cannot be tolerated. The advantage of synchronous replication is that it can maintain high consistency.

Asynchronous replication can guarantee higher system performance, but it may cause inconsistencies. The content can only be written in the master node, and the proxy node can only accept read operations. The content written to the main is indirectly written to one or more proxies. This method is Main/Proxy (M/P), which is an asynchronous replication method. This M/P method provides lower latency and reduces the hit rate. Multi-Main (MM) replication supports writing from multiple nodes at the same time, which leads to consistency problems. The best MM can do is eventual consistency. It is easy for MM to deal with failures because every node can accept writes. Two-phase commit (2PC) is a protocol used to establish transactions between various nodes [25]. This approach severely reduces throughput and increases latency. Paxos is also a consensus protocol [26]. Unlike 2PC, Paxos is decentralised. Although Paxos also has high latency, Paxos has a large number of applications in Google. It has huge advantages in data migration.

Multiversion concurrency control (MVCC) is currently the most popular concurrency control implementation in the database field [27]. MVCC ensures the correctness of transactions as much as possible and maximise concurrency. Today's mainstream databases, such as Oracle [28], MySQL [29], and HyPer [30], almost all support MVCC. T. Neumann et al. have achieved full serialisation in their system without the need for Snapshot Isolation (SI) [31]. Cicada is a single-node multicore in-memory transactional database with serialisability [32]. To provide high performance under diverse workloads, Cicada reduces overhead and contention at several levels of the system by leveraging optimistic and MVCC schemes and multiple loosely synchronized clocks while mitigating their drawbacks.

K. Ma et al. researched the in-memory database version recovery technology, using the unit state model to achieve an unlimited review of any revisions [33]. Q. Cai et al. proposed an efficient distributed memory platform that can provide a memory consistency protocol between multiple distributed nodes [34].

However, all these multi-version schemes are aimed at distributed systems, and there is no optimization method for single-machine multi-core at present. To this end, we propose a replication method of main-proxy backup specifically for multi-core according to the operating characteristics and data structure of our system. This method occupies less hardware resources, and can guarantee system operating efficiency at the expense of a certain amount of real-time performance. This design method fully meets the requirements of our network traffic capture system for the cache system.

3. Model Description

In this section, we design a multicore cache design scheme, which includes caching replacement algorithm, cache setting method, and cache synchronization method.

3.1. System Structure

We designed and implemented a full packet capture (FPC) system [35]. The system has multiple functions, such as packet receiving, nanosecond timestamp, load balancing, data packet preprocessing, application layer protocol analysis, data packet storage, and log management. As shown in Figure 1, there are two processes of auditor and FPC (Full Packet Capture) in Data Plane Development Kit (DPDK) [36]. The auditor process is mainly responsible for data packet capture and preprocessing, and the FPC process is responsible for in-depth processing of data packets. The whole system uses DPDK as a framework to process data packets, and uses a lot of optimization techniques in DPDK. After the network packet is received, the timestamp and packet information are first added to the inter-frame gap. Then, the data packets are load-balanced and distributed to multiple queues. After the data packet is transferred to the DPDK platform, it is copied. One is simply parsed in the FPC process and stored in the hard disk, and the other is transmitted to the high-level protocol parser module in the auditor process for complete parsing. The parsed information is stored in the form of logs. Each task is assigned one or more CPU

core according to the task complexity. The more complex the task, the more cores assigned. For example, the application layer protocol-parsing module is allocated four CPU cores, and the data-packet storage module is allocated eight CPU cores.

Figure 1. System design diagram.

In this paper, we focus on parsing the caching methods in the system, for which we design a new multicore caching model. The area marked in red in the Figure 1 is the cache model. We deploy a proxy cache table on each parsing core and set up a main cache table on the cache core.

The protocol-parsing module writes cache entries in the main table and reads cache entries from the proxy table. After the main cache table is updated and written, it synchronizes the updated information to the proxy table to keep the proxy table in real-time. This read and write isolation is used to reduce the use of locks. In addition, lock usage is the most performance-impacting factor.

To reduce the synchronization overhead further, we split the cache table into sub-tables. When the main cache table is synchronized to the proxy cache table, only those sub-tables that have just been updated need to be synchronized. If the sub-table was not updated in the previous sync cycle, then this sub-table does not need to be synced.

3.2. Cache Set Method

The cache set method is to split the cache items according to their sizes. That is, cache items of similar size are stored in a sub-table, and each sub-table is differentiated by cache item size. As the memory space occupied by the cache items in each sub-table is allocated in advance, this can minimise memory waste. Ideally, cache items of the same size should be grouped together and evenly spaced.

L. Breslau et al. [37] indicate an independent request stream that follows a similar Zipf distribution is sufficient to fit the actual web request law. To get closer to the distribution of DNS data in the network, we use a more refined split method. According to the statistics in the Verisign [38] domain name database, the domain name length distribution is shown in the Figures 2 and 3. The number of shorter domain names is significantly higher; the shorter the domain name, the easier it is for users to remember and understand. Such domain names are used more frequently. Shorter domain names on the internet are also visited more frequently.

Figure 2. Distribution of lengths of all the registered .com domains.

Figure 3. Distribution of lengths of all the registered .net domains.

Therefore, during the initialisation of the sub-cache table, more space is allocated to those sub-tables that store shorter domain names. For example, 50 MB of memory is allocated for a sub-table for storing domain names of 3-bytes long, but 1 MB of memory is allocated for a sub-table for storing a domain name 65-bytes long.

As shown in Figure 4, the system allocates memory blocks uniformly during initialisation to improve memory utilisation, and each memory block stores a sub-cache table. Each sub-cache table has a separate linked list, which is the sub-linked list.

Figure 4. Schematic diagram of cache table split.

As shown in Figure 5, we use the hash table to locate the cache item, so that it can be found and moved. The hash table uses a linked list to resolve hash conflicts. When multiple cache items are hashed to the same key, pointers connect the cache items. The entire cache system needs to maintain a huge hash table to perform a global search on cache items. When the hash table is updated, only the hash bucket is locked to reduce the negative impact of the lock.

Figure 5. LRU data structures.

When moving, adding, or deleting cache items, a linked list is used to perform operations. When locking the linked list, only the entire linked list can be locked. Thus, the locking operation has a huge impact on the linked list. To avoid the performance degradation caused by lock operations, we split the cache table into sub-cache tables so that only a single sub-cache table needs to be locked for each operation. Each sub-linked list is included in the cache sub-table, so that the cache table can only be synchronized with updated sub-tables during synchronization.

3.3. Cache Replacement Algorithm

We have designed a hybrid replacement algorithm that mixes FIFO and LRU. This algorithm combines the advantages of LRU and FIFO buffers, reducing the use of locks as much as possible. Each cache sub-table is divided into three partitions: the hot partition, the temporary storage partition, and the recycle partition. FIFO rules are used internally in hot partitions, which follow the first-in-first-out principle.

When the cache item is written to the cache table for the first time, the cache item is first written to the temporary storage partition. In addition, the flag bit of the cache item is set to 0. If the cache item in temporary partition is hit again, the flag bit of the cache item is set to 1. If the cache item cannot be found in the cache table at this time, the operation is performed the same way as the first time the cache item is written. We describe our implementation of the replacement algorithm in Algorithm 1.

Algorithm 1 Cache replacement

Input: item, command;

Output: item.

```

1:  $flag\_item = 0$ 
2: function CACHE_REPLACEMENT(item, command)
3:   if write then
4:     if item in cache table then
5:       if item in hot partition then
6:         return
7:       else if item in temporary partition then
8:         if  $flag\_item == 0$  then
9:            $flag\_item = 1$ 
10:        else
11:          remove item from temporary partition
12:          insert item to hot partition
13:        end if
14:      else if item in recycle partition then
15:        remove item from recycle partition
16:        insert item to hot partition
17:      end if
18:    else
19:      insert item to temporary partition
20:    end if
21:  else if read then
22:    if item in cache table then
23:      if item in hot partition then
24:        Return item
25:      else if item in temporary partition then
26:        remove item from temporary partition
27:        insert item to hot partition
28:      else if item in recycle partition then
29:        remove item from recycle partition
30:        insert item to hot partition
31:      end if
32:    end if
33:    return item
34:  end if
35: end function
  
```

The three partitions in the cache table are used to store different cache items. At the same time, a flag bit is set to indicate whether the cache item enters the current partition for the first time. In this situation, 0 means that the partition is entered for the first time, and 1 means that the cache item has been hit after entering the partition.

When the item is cached for the first time, it will be stored in the temporary partition. When it is hit for the second time, only the flag is set to 1. The cache item is moved to the hot partition on the third hit. If it has not been hit in the temporary partition, the cache entry is deleted. No operation is performed when the cache item in the hot partition is hit again. When the hot partition is full, the last cache item is moved to the recycle partition instead of deleted directly. The recycling partition is used to store cache items eliminated

by the hot partition temporarily. When the cache item in the reclaimed partition is hit again, the cache item is returned to the hot partition. However, when the cache items in the recycling partition are eliminated, they are eliminated directly. This ensures that the cache items in the hot partition cannot be easily eliminated, and the hit rate of the cache table is guaranteed. In the temporary partition, the cache item is hit three times before being moved to the hot partition. This is a separate design for the DNS request-response parsing scenario. It is also possible to change the cache item in the temporary partition to be moved to the hot partition after being hit twice. The hit mentioned in our article can be either a read operation or a write operation.

3.4. Cache Synchronization Method

As shown in the Figure 6, our method uses a main-proxy replication model. Asynchronous methods are used to synchronize messages, which may reduce the hit rate. The protocol parsing module writes cache entries in the main table and reads cache entries from the proxy table. The protocol parser module reads the proxy cache table on the self core when parsing network traffic. This read is done without locking the proxy cache table, but read indication is sent to the main cache table synchronously. The main cache table adjusts the distribution of the accessed cache items based on this read operation and combined with our proposed cache replacement algorithm.

Figure 6. One-main and multi-proxy synchronization method.

When the protocol parser module parses the traffic and obtains the information it needs to store, it writes the cache entries to the main cache table. This write operation requires locking because it is a multicore operation. In this way, the main cache table updates the read operations and write operations. The main cache table is adjusted the cache table according to the cache replacement algorithm based on all these operations.

The proxy cache table working on the parsing core is not updated in real time, and only the read indication is forwarded to the main cache table. This design can avoid frequent lock operations caused by cache replacement and can ensure high throughput and low system latency. At the same time, the main cache table accepts write operation instructions and read operation instructions, and the cache table must be replaced and updated according to these instructions.

Because the cache table is split into cache sub-tables, we only need to synchronize the sub-tables whose information has changed during synchronization. The sub-tables with the same information do not need to be synchronized. This design greatly reduces the amount of data that needs to be synchronized each time. The data flows from the parsing module to the main cache table and then to the proxy cache table. The flow direction of the data flow is one-way. When synchronizing data, it can only be synchronized from the main cache table to the proxy cache table. Therefore, there are no data inconsistencies.

The main cache table periodically (for example, every second) synchronizes data to the proxy cache table. After receiving the synchronization information, the proxy cache table starts to update its own cache table when it is not busy.

4. Evaluation

In this section, we evaluate the performance of the system through simulation. We performed simulation experiments on our method using trace-driven simulation. The emulator requests the domain name obtained from the log file. When a new request arrives, the cache system retrieves the cached content according to the key to confirm whether the corresponding content already exists in the cache. If it is, the cache table remains unchanged, and the hit counter of the document is increased by one. Otherwise, assuming that the cache is not full, the cache item corresponding to the request is stored in the cache table. When the cache is full, certain cache items are deleted from the cache according to a predefined replacement strategy.

The server is PowerEdge R720xd with 2 Intel Xeon E5-2609 @ 2.40GHz, 16GB DDR4 memory. The system is Centos 7.4 with 64-bit Linux 3.10.0 kernel.

4.1. Insertion Latency

A network-traffic capture and analysis system is very sensitive to the latency of the caching system. Therefore, we need to evaluate the latency of the system. For caching systems, the insertion operation is usually the most time-consuming. Therefore, we are wired to evaluate the insertion latency of the cache system.

We test LRU, FIFO, and our proposed hybrid culling method and compare their performance under single-core and multicore. All configurations are the same except that the three retirement algorithms are different. Without specifying, the multicore configuration method uses our proposed method of primary and proxy caching, and the synchronization mechanism uses our proposed periodic synchronization method. We tested the performance of the three retirement methods under different load factors. Each point is the average value derived after ten experiments.

As shown in Figure 7, under single-core conditions, LRU buffer latency is higher than FIFO and our method. In addition, our method has almost the same latency as FIFO. The load factor is a variable that symbolises the amount of cached content. As the amount of cached content continues to increase, the latency of all three caching methods increases. However, the FIFO and our method are increasing slowly. Under single-core conditions, when the load factor reaches 100%, the latency of the LRU method reaches 1198 ns, but the latency of the FIFO and our method are 143 ns and 167 ns, respectively.

Figure 7. Insert latency of different cache replacement algorithms under different load factors in the single-core.

As shown in Figure 8, when four cores are running in parallel, the latency of the three caching methods is significantly higher. The LRU insert latency increases from 1198 ns to 5991 ns. The FIFO insert latency increases from 143 ns to 165 ns, and the insert latency of our method increases from 167 ns to 324 ns. From the perspective of the increase ratio, the LRU insert latency increases tremendously, and the multicore latency is almost six times that of the single-core latency. The latency increase of FIFO and our method is relatively limited.

Figure 8. Insert latency of different cache replacement algorithms under different load factors in the quad-core.

As shown in Figure 9, under the eight-core condition, the LRU insert latency increases to 141,181 ns. The FIFO and our method only increases to 1688 ns and 3636 ns. This shows that, as the number of cores running in parallel increases, the latency of LRU increases rapidly, but the increases in FIFO and our method are not obvious. Therefore, the LRU algorithm is not suitable for multicore parallel operation, and our method and FIFO method perform well on multicore systems.

Figure 9. Insert latency of different cache replacement algorithms under different load factors in the eight-core.

4.2. Throughput

Throughput is a commonly used indicator of network equipment. To evaluate the throughput performance of our caching method, we conduct experiments on the three caching methods. The experiment set up two modes 100% put, and 30% put operation + 70% get operation, respectively. Each data is the result of 10 averaged experiments.

As shown in Figure 10, the throughput of FIFO and our method are significantly higher than the LRU throughput, and the throughput of FIFO and our method increase faster with the number of cores. Figure 11 shows that, when the 30% put operation is added, the cache speed improves. We think this is because these two methods respond faster to put operations and use less computing resources.

Figure 10. Throughput at 100% put operation.

Figure 11. Throughput at 70% put operation and 30% get operation.

We did experiments from one to eight cores, and we tested each result ten times and averaged. Figure 12 show that, as the number of cores running in parallel increases, the throughput per core continues to decrease. The throughput of a single core is significantly higher than the throughput of a multicore. When the number of cores increases, for example the number of cores exceeds four the throughput per core decreases slowly. The throughput of FIFO and our method perform better when there are get operations. We think this is because FIFO and our method do not need to move the cache items in some get operations, thus saving computing resources.

Figure 12. The average throughput of each core with different number of cores.

4.3. Hit Ratio

The hit rate is also a very important indicator for the cache system. If the hit rate is too low, the cache system can hardly work. To increase the hit rate, the cache usually stores as many smaller cache items as possible. The total available size of the cache system is 8 GB. We used nine different cache sizes to test the hit rate of the cache method. We conducted experiments on LRU, FIFO, and our method to evaluate the hit rate of the cache replacement algorithm. The total size of the unique domain name used in the test is 17 GB. The domain name is distributed according to Ziff to test the cache system (90% get and 10 set).

Figure 13 shows the hit rate of each cache replacement method under different cache sizes. For any cache replacement method, an increase in cache size always increases the hit rate. In terms of hit rate, LRU and our method obviously have better performance levels than FIFO. In contrast, the FIFO has the worst hit rate because there are a large number of duplicate buffer items in the FIFO. This significantly wastes a large portion of cache space. The hit rate of LRU is slightly higher than the hit rate of our method, which may be because some popular caches are replaced with recycle partitions and do not return to the popular partitions in time. This leads to a miss when this popular cache item is accessed again.

Figure 13. Hit rate under different cache sizes.

4.4. Synchronous Method Test

To test the throughput and the hit rate of different synchronization methods, we conducted experimental tests on three methods: synchronous, asynchronous, and non-synchronous. The way of synchronization means that each operation of each cached sub-table is immediately synchronized to all other sub-tables and the total table. The asynchronous mode refers to the periodic synchronization strategy we propose. Non-synchronization means that each sub-table operates independently on its own core and does not need to be synchronized to any other sub-tables. This is equivalent to a single-core operation. We use our cache replacement method in the experiment. In the experiment, the total size of the cache is 8 GB, and the cache is tested with 17 GB of unique domain names.

Table 1 shows the insert latency and hit rate of three different synchronization methods. As can be seen from Table 1, the one main multi-proxy approach has a much higher hit rate than the single operation approach does. However, the single operation method has a high throughput. Except when the cache space is sufficient, the asynchronous method is better than the single operation method. Compared with the asynchronous method, the synchronous method has no obvious advantage in the hit rate, but the throughput is very low. Therefore, we believe that our periodic asynchronous synchronization method is the best method.

Table 1. Performance comparison of the three versions of synchronization methods. The three methods are the one-main multi-proxy real-time synchronization method, the one-master multi-agent asynchronous synchronization method, and the method where each operation is synchronized separately.

One Main, Multiple Proxy				Single Operation	
Synchronous		Asynchronous			
throughput	Hit ratio	throughput	Hit ratio	throughput	Hit ratio
10691274	68.04	27460288	68.38	75897206	14.32

4.5. Comparison with Other Methods

The cache system we designed is only a module in the full packet capture system. The cache systems in Table 2 are all dedicated and complete cache systems. It is very difficult and unrealistic to transplant these systems into our systems and test them. Therefore, we only briefly compared the results of these methods with our methods. We know that the get operation occupies less computing resources compared with the insert operation. The best-performing MICA in the cache system performs best in the case of 50% get operations, and the corresponding throughput is 32.8 Mops. Our caching system achieved a throughput of 27.2 Mops with 30% of get operations. This fully proves that our cache system can

guarantee extremely high throughput performance when inset operations account for a relatively high proportion (70%).

Table 2. The throughput of our caching system relative to other memory caching systems.

Method	Get Ratio	Cores	Throughput (Mops)
our	30%	8	27.2
MICA [23]	50%	8	32.8
MEMC3 [21]	85%	8	4.8
LaKe [39]	100%	FPGA	13.1
Masstree [6]	100%	16	9.93
Cavast [40]	0%	6	4.2

5. Conclusions

In this paper, a key-value pair caching method including cache replacement algorithm, cache setting and cache synchronization method is proposed for multiple core applications. Through copying only a small number of sub-tables at a time, the periodic asynchronous synchronization method we proposed can significantly reduce the use of locks, which is the most time-consuming operation in the cache system. The proposed cache replacement algorithm combines the advantages of both FIFO and LRU. There is almost no performance gap between it and the best performance in each test. Compared with other methods, our caching system achieved a throughput of 27.2 Mops with 30% of get operations. This fully proves that our cache system can guarantee extremely high throughput performance when insert operations account for a relatively high percentage (70%). In summary, the proposed cache system perfectly fulfills our requirements, and can be transplanted to other multicore scenarios through simple modifications.

Author Contributions: Methodology, L.H.; software, L.H.; validation, L.H.; resources, Z.G.; writing—original draft preparation, L.H.; writing—review and editing, Z.G. and X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Strategic Science and Technology Pilot Project of the Chinese Academy of Sciences: SEANET Technology Standardization Research and System Development OF FUNDER grant number Y929011611.

Data Availability Statement: The data set generated or analyzed during the current research can be obtained from the corresponding author according to reasonable requirements.

Acknowledgments: The authors would like to thank Lei Liu, Lei Song, Chuanhong Li, and Ce Zeng for insightful comments. The authors would like to sincerely thank the anonymous reviewers for their feedback on earlier versions of this manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Atikoglu, B.; Xu, Y.; Frachtenberg, E.; Jiang, S.; Paleczny, M. Workload analysis of a large-scale key-value store. In Proceedings of the 12th ACM Sigmetrics/Performance Joint International Conference on Measurement and Modeling of Computer Systems, London, UK, 11–15 June 2012; pp. 53–64.
- Marathe, V.J.; Seltzer, M.; Byan, S.; Harris, T. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In Proceedings of the 9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17), Santa Clara, CA, USA, 10–11 July 2017.
- Kim, H.; Rajkumar, R. Real-time cache management for multi-core virtualization. In Proceedings of the 2016 International Conference on Embedded Software (EMSOFT), Pittsburgh, PA, USA, 2–7 October 2016; pp. 1–10.
- Blankstein, A.; Sen, S.; Freedman, M.J. Hyperbolic caching: Flexible caching for web applications. In Proceedings of the 2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17), Santa Clara, CA, USA, 12–14 July 2017; pp. 499–511.
- DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Voshall, P.; Vogels, W. Dynamo: Amazon’s highly available key-value store. *ACM Sigops Oper. Syst. Rev.* **2007**, *41*, 205–220. [[CrossRef](#)]

6. Mao, Y.; Kohler, E.; Morris, R.T. Cache craftiness for fast multicore key-value storage. In Proceedings of the 7th ACM European Conference on Computer Systems, Bern, Switzerland, 10–13 April 2012; pp. 183–196.
7. Wu, F.; Yang, M.H.; Zhang, B.; Du, D.H. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In Proceedings of the 2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20), Virtual, Online, 15–17 July 2020; pp. 603–615.
8. Tokusashi, Y.; Matsutani, H. A multilevel NOSQL cache design combining in-NIC and in-kernel caches. In Proceedings of the 2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI), Santa Clara, CA, USA, 24–26 August 2016; pp. 60–67.
9. Jin, X.; Li, X.; Zhang, H.; Soulé, R.; Lee, J.; Foster, N.; Kim, C.; Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; pp. 121–136.
10. Wu, X.; Zhang, L.; Wang, Y.; Ren, Y.; Hack, M.; Jiang, S. zexpander: A key-value cache with both high performance and fewer misses. In Proceedings of the Eleventh European Conference on Computer Systems, London, UK, 18–21 April 2016; pp. 1–15.
11. Chen, Y.; Lu, Y.; Yang, F.; Wang, Q.; Wang, Y.; Shu, J. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 16–20 March 2020; pp. 1077–1091.
12. Eisenman, A.; Cidon, A.; Pergament, E.; Haimovich, O.; Stutsman, R.; Alizadeh, M.; Katti, S. Flashield: A key-value cache that minimizes writes to flash. *arXiv* **2017**, arXiv:1702.02588.
13. Chen, L.; Li, J.; Ma, R.; Guan, H.; Jacobsen, H.A. EnclaveCache: A secure and scalable key-value cache in multi-tenant clouds using Intel SGX. In Proceedings of the 20th International Middleware Conference, Davis, CA, USA, 9–13 December 2019; pp. 14–27.
14. Yang, J.; Yue, Y.; Rashmi, K. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In Proceedings of the 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20), Banff, AB, Canada, 4–6 November 2020; pp. 191–208.
15. Wang, Y.; Yang, Y.; Han, C.; Ye, L.; Ke, Y.; Wang, Q. LR-LRU: A PACS-Oriented intelligent cache replacement policy. *IEEE Access* **2019**, *7*, 58073–58084. [[CrossRef](#)]
16. Waldspurger, C.; Saemundsson, T.; Ahmad, I.; Park, N. Cache modeling and optimization using miniature simulations. In Proceedings of the 2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17), Santa Clara, CA, USA, 12–14 July 2017; pp. 487–498.
17. Shen, Z.; Chen, F.; Jia, Y.; Shao, Z. Didacache: An integration of device and application for flash-based key-value caching. *ACM Trans. Storage (TOS)* **2018**, *14*, 1–32. [[CrossRef](#)]
18. Jia, Y.; Shao, Z.; Chen, F. SlimCache: Exploiting data compression opportunities in flash-based key-value caching. In Proceedings of the 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, 9–11 September 2018; pp. 209–222.
19. Idreos, S.; Callaghan, M. Key-Value Storage Engines. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 2667–2672.
20. Xing, Y.; Liu, F.; Xiao, N.; Chen, Z.; Lu, Y. Capability for multi-core and many-core memory systems: A case-study with xeon processors. *IEEE Access* **2018**, *7*, 47655–47662. [[CrossRef](#)]
21. Fan, B.; Andersen, D.G.; Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In Proceedings of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13), Lombard, IL, USA, 2–5 April 2013; pp. 371–384.
22. Chandramouli, B.; Prasaad, G.; Kossmann, D.; Levandoski, J.; Hunter, J.; Barnett, M. Faster: A concurrent key-value store with in-place updates. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 275–290.
23. Lim, H.; Han, D.; Andersen, D.G.; Kaminsky, M. {MICA}: A holistic approach to fast in-memory key-value storage. In Proceedings of the 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), Seattle, WA, USA, 2–4 April 2014; pp. 429–444.
24. Metreveli, Z.; Zeldovich, N.; Kaashoek, M.F. Cphash: A cache-partitioned hash table. *ACM Sigplan Not.* **2012**, *47*, 319–320. [[CrossRef](#)]
25. Lamport, B.; Lomet, D. A new presumed commit optimization for two phase commit. In Proceedings of the 19th VLDB Conference, Dublin, Ireland, 24–27 August 1993.
26. Lamport, L. Paxos made simple. *ACM Sigact News* **2001**, *32*, 18–25.
27. Wu, Y.; Arulraj, J.; Lin, J.; Xian, R.; Pavlo, A. An empirical evaluation of in-memory multi-version concurrency control. *Proc. Vldb Endow.* **2017**, *10*, 781–792. [[CrossRef](#)]
28. Lahiri, T.; Chavan, S.; Colgan, M.; Das, D.; Ganesh, A.; Gleeson, M.; Hase, S.; Holloway, A.; Kamp, J.; Lee, T.H.; et al. Oracle database in-memory: A dual format in-memory database. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 1253–1258.
29. Wikipedia Contributors. MySQL—Wikipedia, The Free Encyclopedia. 2021. Available online: <https://en.wikipedia.org/wiki/MySQL> (accessed on 23 June 2021).

30. Kemper, A.; Neumann, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, Washington, DC, USA, 11–16 April 2011; pp. 195–206.
31. Neumann, T.; Mühlbauer, T.; Kemper, A. Fast serializable multi-version concurrency control for main-memory database systems. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, VIC, Australia, 31 May–4 June 2015; pp. 677–689.
32. Lim, H.; Kaminsky, M.; Andersen, D.G. Cicada: Dependably fast multi-core in-memory transactions. In Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, IL, USA, 14–19 May 2017; pp. 21–35.
33. Ma, K.; Yang, B. Stream-based live data replication approach of in-memory cache. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e4052. [\[CrossRef\]](#)
34. Cai, Q.; Guo, W.; Zhang, H.; Agrawal, D.; Chen, G.; Ooi, B.C.; Tan, K.L.; Teo, Y.M.; Wang, S. Efficient distributed memory management with RDMA and caching. *Proc. Vldb Endow.* **2018**, *11*, 1604–1617. [\[CrossRef\]](#)
35. Han, L.; Guo, Z.; Huang, X.; Zeng, X. A Multifunctional Full-Packet Capture and Network Measurement System Supporting Nanosecond Timestamp and Real-Time Analysis. *IEEE Trans. Instrum. Meas.* **2021**, *70*, 1–12.
36. Zeng, L.; Ye, X.; Wang, L. Survey of Research on DPDK Technology Application. *J. Netw. New Media* **2020**, *9*, 1–8.
37. Breslau, L.; Cao, P.; Fan, L.; Phillips, G.; Shenker, S. Web caching and Zipf-like distributions: Evidence and implications. In Proceedings of the IEEE INFOCOM '99. Conference on Computer Communications, Proceedings, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, The Future is Now (Cat. No.99CH36320), New York, NY, USA, 21–25 March 1999; Volume 1, pp. 126–134.
38. Wikipedia Contributors. Verisign—Wikipedia, The Free Encyclopedia. 2021. Available online: <https://en.wikipedia.org/wiki/Verisign> (accessed on 23 June 2021).
39. Tokusashi, Y.; Matsutani, H.; Zilberman, N. Lake: An energy efficient, low latency, accelerated key-value store. *arXiv* **2018**, arXiv:1805.11344.
40. Wang, K.; Liu, J.; Chen, F. Put an elephant into a fridge: Optimizing cache efficiency for in-memory key-value stores. *Proc. Vldb Endow.* **2020**, *13*, 1540–1554. [\[CrossRef\]](#)