GÜFBED/*GUSTIJ* (2020) 10 (3): 844-852 **Gümüşhane Üniversitesi Fen Bilimleri Enstitüsü Dergisi**
DOI: 10.17714/gumusfenbil.637942 Araştırma Makalesi / *Research Article*

A Performance Comparison of Java Cache Memory Implementations

Java Ön Bellek Gerçekleştirmelerinin Bir Performans Karşılaştırması

Abdullah Talha KABAKUŞ*

Düzce Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, 81620, Düzce, Türkiye

• Geliş tarihi / *Received*: 24.10.2019 • Düzeltilerek geliş tarihi / *Received in revised form*: 21.06.2020 • Kabul tarihi / *Accepted*: 23.06.2020

Abstract

Nowadays the information systems are substantially data-intensive and the data is going to be more critical than before. For these systems, which are intolerant in terms of time latency, the way of accessing data becomes more critical. In these situations, an additional data layer named cache memory is used. There are various both open-source and commercial Java cache memory implementations based on the specifications defined by Java Community Process. In this study, the most widely used Java cache memory implementations are evaluated in order to compare their performances in terms of elapsed time and memory consumption. The experimental results imply that the architectural design of cache memory has a great effect on performance and there is no winner that provides the best performance for all data operations.

Keywords: Cache, Cache Memory, Caching, Data, Java

Öz

Günümüzde bilgi sistemleri ciddi miktarda veri ağırlıklıdır ve bu veri gitgide daha kritik hale gelmektedir. Bu tip veriye erişimde gecikmeye toleransı olmayan sistemlerde, veriye ulaşım yöntemi çok daha kritik hale gelmektedir. Bu tip durumlarda, ön bellek hafızası ismi verilen ek bir veri katmanı kullanılmaktadır. Java Community Process tarafından tanımlanan şartlara uyan gerek açık kaynak kodlu gerekse ticari çeşitli Java ön bellek gerçekleştirmeleri mevcuttur. Bu çalışmada, en çok kullanılan Java ön bellek gerçekleştirmeleri ihtiyaç duyduğu zaman ve bellek tüketim açısından performans karşılaştırması yapmak üzere değerlendirilmiştir. Deneysel sonuçlar ön bellek mimari tasarımının performans üzerine ciddi etkisi olduğunu ve tüm veri işlemleri için en iyi performansı gösteren tek bir kazananın olmadığını göstermiştir.

***Anahtar kelimeler**: Ön Bellek, Ön Bellek Hafızası, Ön Bellekleme, Veri, Java*

*Abdullah Talha KABAKUŞ; talhakabakus@duzce.edu.tr; Tel: (0380) 542 10 36; orcid.org/0000-0003-2181-4292

1. Introduction

With the investment of Web 2.0, the data generated by the online resources has increased enormously since Web 2.0 harnesses the Web in a more interactive, responsive and collaborative way, emphasizing peers' social interaction and collective intelligence, and presents new opportunities for leveraging the Web and engaging its users more effectively (Murugesan, 2007). As a result of this interaction, a huge amount of data is being generated daily by the Web 2.0 services such as social networks and financial markets (Jose et al., 2011). According to a recent report, while *Facebook* stores, accesses, and analyzes 30 + Petabytes of user-generated data with the increase of 100 terabytes of data uploaded daily, *YouTube* users upload 48 hours of new video every minute of the day (Mark Mulcahy, 2017). A requirement of the interaction between the browser and the end-user, which is introduced by Web 2.0, is processing the data quickly in order to provide lower latency response times (Carra and Michiardi, 2014). The traditional data-intensive information systems tend to use a database management system in order to store their data on persistent storage such as hard disks. The idea of using cache memories comes from that an additional layer, which stores the most frequently/recently used data on a memory area which is much faster than random access memory (RAM), could only improve the speed to access the data and bridge the performance gap between processor and RAM (Swain et al., 2018). For this reason, fetching data from cache plays an important role in increasing system performance (Akbari Bengar et al., 2020). Therefore, modern information systems, which are data-intensive in terms of their business model, tend to use cache memories alongside the persistent storage. In a similar fashion, processing in memory (PIM) has been used with in-memory computing for processing large data-intensive applications such as machine learning, graph processing, social network analysis, and image processing (Ahn et al., 2015; Chi et al., 2016; Fattahi et al., 2019; Martins et al., 2017; Nai et al., 2017). In addition to providing faster access to the data, cache memory also plays a critical role in power reduction (Panda et al., 2016). There exist various both open-source or commercial cache memory implementations. In this study, the most widely used Java cache memory implementations are evaluated in order to shed light on their performances in terms of both (1) the amount of memory used to complete data operations, and (2) the elapsed time to complete each operation. The operations were determined as (1) querying the cache for entry of a given key, (2) adding a new entry into the cache, (3) removing an entry from the cache through the given key, and (4) getting whole available data from the cache. The rest of the paper is structured as follows: Section 2 introduces the cache memory implementations used in this study. Section 3 presents the experimental results and discussion. Finally, Section 4 concludes the paper by summarizing the findings with directions for future work.

2. Java Cache Implementations

Java Specification Request (JSR)-107, also referred to as *JCache*, is a specification that defines *javax.cache* API and semantics for temporary, in-memory caching of Java objects (URL-1, 2017). There are some both open-source and commercial implementations of *JSR-107*. These implementations use different eviction algorithms that are used when the cache is full in terms of storage and a new entry is needed to be inserted into the cache. In the following subsections, the most widely used Java cache memory implementations based on *JSR-107*, which are also used in this study, are briefly introduced.

2.1. EhCache

Ehcache is an open-source implementation of the *JSR-107*, which is reported to be the most used Java-based cache (URL-2, 2020). *Ehcache* uses the least recently used (LRU) algorithm to insert a new entry when the cache is full in terms of storage. While the LRU eviction algorithm removes the least recently used entities from the cache, the least frequently used (LFU) algorithm removes the least frequently used entities from the cache when the cache is full in terms of storage. The LRU algorithm assumes the cache line, which is used least in the recent past, is used in the near future with the least probability (Yeung and Ng, 1997).

2.2. Guava

Guava is the Java cache implementation provided by *Google*. Similar to *Ehcache*, *Guava* also uses LRU as the eviction algorithm. *Guava* provides various collection types, a graph library, functional types, APIs for concurrency, input/output, hashing alongside in-memory cache (URL-3, 2020).

2.3. Cache2K

Cache2K is a high-performance Java cache that serves inside Java virtual machine (JVM). *Cache2K* uses a modern eviction algorithm, that utilizes both the recency and frequency aspects, which is also referred to as Least Recently Frequently Used (LRFU) (Alghazo et al., 2004). The LFRU algorithm associates each block in the cache with a value called Combined Recency and Frequency (CRF) which is used to exploit temporal locality and reference popularity (Bahn and Noh, 2003). Then it replaces the block in the cache with the minimum CRF value (Donghee Lee et al., 2001; Hennessy and Patterson, 1998; Jinhyuk Yoon et al., 2002; Lee et al., 1999; Wang et al., 2002; Wong and Baer, 2000). According to the various performance experiments, *Cache2K* is reported as one of the fastest caches available for Java (URL-4, 2020). *Cache2K* utilizes the LFU eviction algorithm, which uses the history of accesses to predict the probability of a subsequent reference (Yeung and Ng, 1997) alongside the LRU eviction algorithm.

2.4. Memcached

Memcached is an open-source, high-performance, distributed memory object cache. Similar to *Ehcache* and *Guava*, *Memcached* uses LRU as the eviction algorithm. *Memcached* is commonly used to speed up dynamic web applications by alleviating the database load (Fitzpatrick, 2004). *Facebook* is one of the most popular users of *Memcached* as it was reported in 2008 that *Facebook* had 800 *Memcached* servers with up to 28 Terabytes of data in their cache (Hoff, 2009). One of the biggest advantages of *Memcached* is that it is platform-independent and scalable since the clients connect to the cache via sockets (Petrovič, 2008). Thanks to this architectural design, *Memcached* is able to combine the performance of message-passing systems and the simplicity of distributed shared memory.

2.5. Ignite

Ignite is the in-memory computing platform provided by *Apache* which is durable, strongly consistent, and highly available. *Ignite* uses both LRU and FIFO (First In, First Out) as the eviction algorithms. Despite serving from the memory, *Ignite* contains some differences from NoSQL (Not Only SQL) databases such as (1) *Ignite* supports SQL (Structured Query Language), (2) *Ignite* supports collocated processing, and (3) *Ignite* provides strong consistency while NoSQL databases provide eventual consistency (URL-5, 2020).

2.6. Hazelcast

Hazelcast is an open-source, in-memory distributed data grid based on Java. *Hazelcast* should not be considered as purely a cache as it supports a number of distributed collections and features such as specialized collections, concurrency utilities, and listeners (Johns, 2015). *Hazelcast* can be configured to use both LRU and LFU as the eviction algorithm.

While *Cache2K*, *Ehcache*, *Guava*, and *Ignite* directly service from the source code, *Hazelcast*, and *Memcached* require an external process running on the operating system. The comparison of Java cache memory implementations which are introduced above is listed in Table 1.

Table 1. The comparison of Java cache memory implementations

Cache Memory	Eviction Algorithm	Supports Distributed Architecture?	Directly Service from Source Code
Cache2K	LRU, LFU	No	Yes
Ehcache	LRU	Yes	Yes
Memcached	LRU	Yes	No
Guava	LRU	No	Yes
Ignite	LRU, FIFO	Yes	Yes
Hazelcast	LRU, LFU	Yes	No

3. Experimental Results and Discussion

In order to reveal the performance of cache memories, each implementation was evaluated with four different experiments: (1) Querying the cache for an entity through a given key, (2) inserting a new entry into the cache, (3) deleting an existing entry through a given key, and (4) getting the whole entries available in the cache. The whole experiments were carried out on the

same machine whose hardware and software details are listed in Table 2. The elapsed times to complete experiments were calculated by determining the time interval by retrieving the current time through the $java.lang.System.nanoTime$ method, which returns the current time in the most precise way (in nanoseconds) just before and after each experiment. All cache memories were run in the single-node mode (*a.k.a.* standalone) since not all cache memories support the distributed architecture. Therefore, the effect of the distributed architecture on the performance of cache memory is out of the scope of this study. *The Stack Exchange Data Dump* (URL-6, 2009) was downloaded for the sake of performance comparison of cache memories. Before conducting each designed experiment, the data, which is stored in an XML file, was read and stored in Java collections (e.g., $Maps$). Then, each designed experiment was conducted to reveal the performances of the cache memories for the different data operations. It is worth to mention that all experiments were run for 5 times and the average performance was regarded as the final performance of each cache memory in order to ensure that any other running processes on the CPU has not affected the performance. The block diagram of the proposed approach is presented in Fig. 1.

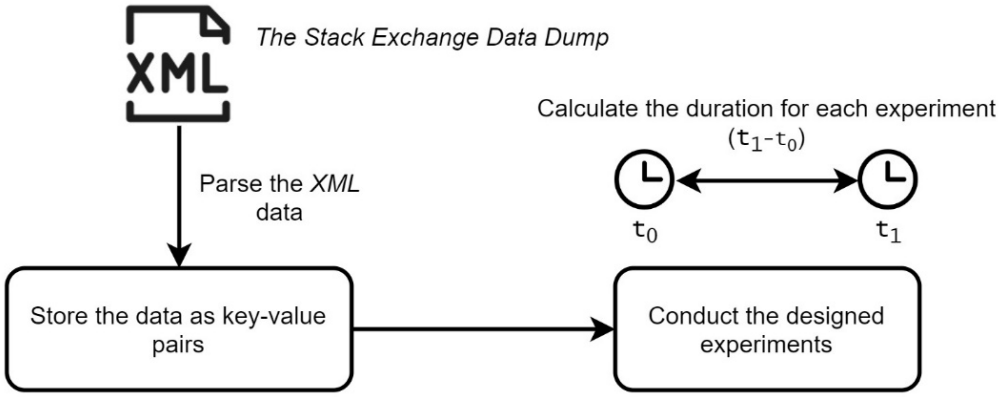

Figure 1. The block diagram of the proposed approach

Table 2. The hardware and software details of the machine which the experiments were carried out on

Operating System	Ubuntu 14.04 (64-bit)
CPU	Intel Core i7-4710MQ 4-Cores; 6 MB L3; 2.50 GHz>3.50 GHz
Memory	16 GB DDR3
Disk	7200 RPM SATA-3
Java Virtual Machine	Oracle Java 1.8.0_121

3.1. Experiment #1 – Querying for an entity through a given key

In order to compare the required time to retrieve an entity, all caches were queried for the same entity through a given key. Before evaluating this experiment, the whole data (see Section 3.2 for more detail) was inserted into the cache memories. For the sake of this experiment, a $user$ was queried through his/her Id, which was randomly chosen from the available set of keys. As the experimental result is presented in Fig. 2, *Cache2K* and *Ehcache* provided the best performance in terms of the elapsed time. *Guava* was found as the worst cache memory as it was able to complete the query 1,000 times fold later than *Cache2K* and *Ehcache*.

3.2. Experiment #2 – Inserting bulk data into the caches

The Stack Exchange Data Dump contains several XML files that represent the data of the *Stack Exchange* platform and the part related to "$users$", which contains 86,110 $users$, was stored in the caches. The size of the dump is 1.15 GB and it contains the profile data of the $users$ of *Stack Exchange*. The attributes of $users$ are Id, $Reputation$, $CreationDate$, $DisplayName$, $LastAccessDate$, $WebsiteUrl$, $Location$, Age, $AboutMe$, $Views$, $UpVotes$, and $DownVotes$. For the sake of experiments, the Id, and $DisplayName$ attributes of each $user$ were stored as "key-value" pairs in the cache memories. This operation is also known as the "*put all*"

operation. As the experimental result is presented in Fig. 3, *Ignite*, *Guava*, and *Cache2K* provided quite better performance in terms of elapsed time to insert the bulk data compared to other caches. *Memcached* was found as the slowest cache memory in terms of writing data.

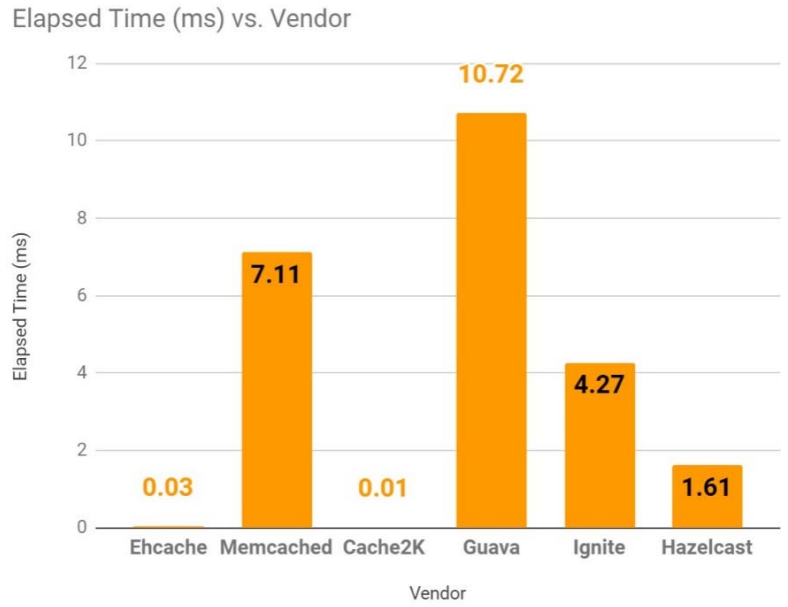

Figure 2. The elapsed times to query the cache through the provided key

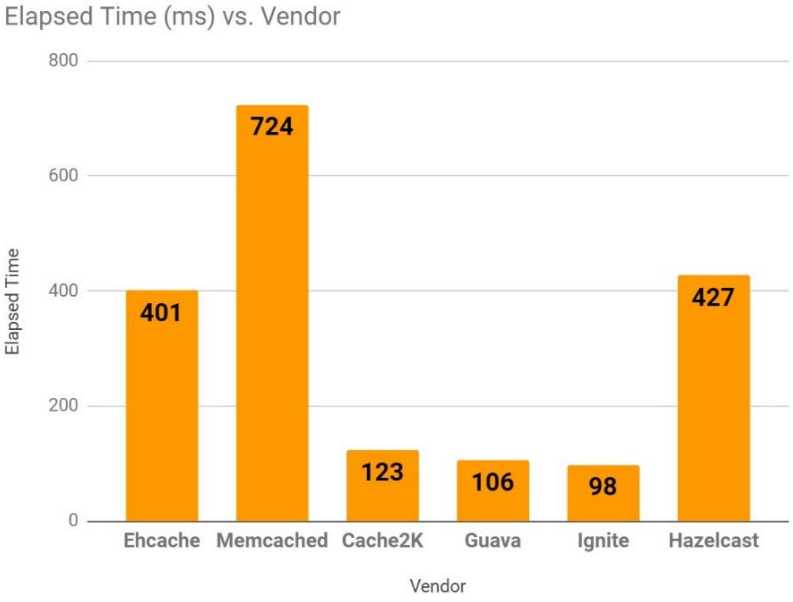

Figure 3. The elapsed times to insert entries into the caches

3.3. Experiment #3 – Deleting an entry from the cache through the given key

In order to compare the entity deletion performances of caches, the same entry was deleted from each cache through its key, which was the Id of the $user$. Similar to *Experiment #1*, the key was randomly chosen from the available set of keys. As the experimental result is presented in Fig. 4, *Cache2K* and *Guava* provided quite better performance in terms of elapsed time compared to the other caches. *Ignite* was found as the slowest cache memory in terms of elapsed time to delete an entry.

3.4. Experiment #4 – Get the whole data from the cache

Despite each cache contains the same data which was the previously inserted 86,110 entries, the elapsed times to get the whole data were calculated quite unusual as the experimental result

is presented in Fig. 5. *Cache2K* was found as the fastest cache memory in terms of reading bulk data from the cache.

The other aspect of the performance evaluation is the memory consumption of each cache memory implementation while storing or manipulating the data. Since all caches stored the same data, the amount of memory each cache consumes can be used to compare the memory usage of caches. The consumed memory was monitored using an open-source tool, namely, *VisualVM* (Sedlacek and Hurka, 2020). As the memory consumptions of the caches are presented in Fig. 6, *Guava* and *Cache2K* were found as the two most efficient cache memories in terms of memory consumption.

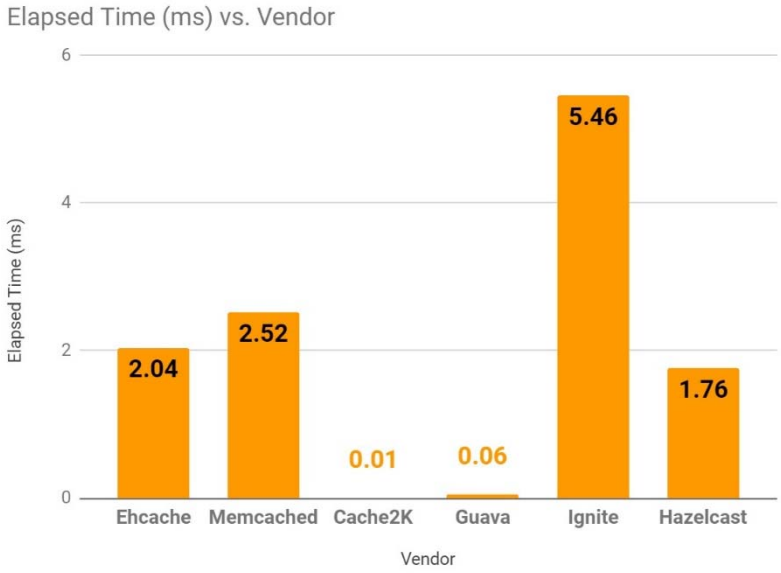

Figure 4. The elapsed times to delete an entry from the cache through the given key

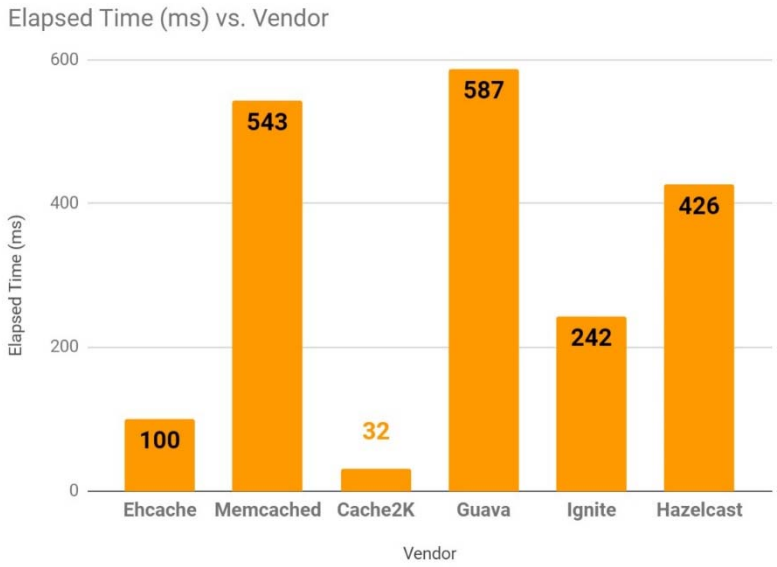

Figure 5. The elapsed times to get all the data available in caches

When all experimental results are reviewed, it is safe to make a conclusion like that *Cache2K* and *Guava* provide the best overall performance in terms of elapsed times to complete operations. The performance differences between the cache memories became more evident when the number of entries that are going to be operated was increased. A limitation of *Memcached* is that it does not provide any methods to get the whole available data. For this operation, the whole data of the cache was retrieved through a loop, which means extra operations that eventually extend the duration of the operation. This limitation could be one of the reasons for *Memcached* for being much slower than most of the cache memories while reading the whole data in the cache. The

limitation of *Cache2K* and *Guava* is that they do not support service in a distributed architecture which could be necessary when the stored data is huge and intolerant in terms of loss. *Guava* and *Cache2K* provided better performance when it comes to efficient memory management. *Ignite* was found as the worst cache memory by consuming 1.8 times more memory than *Guava*.

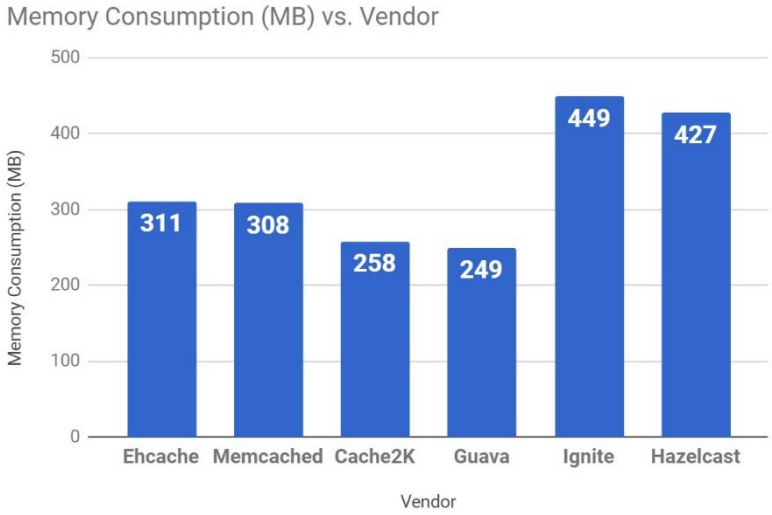

Figure 6. The memory consumptions of cache memories (in MB) to store 86,110 entries

4. Conclusion

Cache memories are commonly used within the data-intensive systems in order to accelerate the speed of the queried data. In this study, several experiments were evaluated in order to reveal their performances in terms of elapsed time to complete each experiment and memory consumption since there are various cache memory implementations. According to the experimental results, the architecture of the cache memory has a great effect on the performance. Additionally, as the experimental result proved, there is no clear winner among the Java cache implementations when it comes to performance. A hybrid of Java cache memories may be used in order to provide the best performance through the different data operations. According to the experimental results, supporting distributed caching comes with an overhead in elapsed time. As future work, the cache memory implementations which are able to service in distributed mode may be evaluated in order to reveal the effect of the data distribution and the distribution strategy on the performance. Also, the architectural design of each evaluated cache memory is needed to be inspected in detail in order to reason the differences between the experimental results. Finally, the cache memories can be evaluated with different eviction scenarios to reveal the performances of cache memories for specific scenarios.

References

Ahn, J., Yoo, S., Mutlu, O., and Choi, K. (2015). PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 2015), 336–348. https://doi.org/10.1145/2749469.2750385

Akbari Bengar, D., Ebrahimnejad, A., Motameni, H., and Golsorkhtabaramiri, M. (2020). A Page Replacement Algorithm based on a Fuzzy Approach to Improve Cache Memory Performance. Soft Computing, 24, 955–963. https://doi.org/10.1007/s00500-019-04624-w

Alghazo, J., Akaaboune, A., and Botros, N. (2004). SF-LRU Cache Replacement Algorithm. Records of the IEEE International Workshop on Memory Technology, Design and Testing, 19–24. https://doi.org/10.1109/MTDT.2004.1327979

Bahn, H., and Noh, S. H. (2003). Characterization of Web Reference Behavior Revisited: Evidence for Dichotomized Cache Management. Information Networking, Networking Technologies for Enhanced Internet Services International Conference 2003 (ICONN 2003), 1018–1027. https://doi.org/10.1007/978-3-540-45235-5_100

Carra, D., and Michiardi, P. (2014). Memory Partitioning in Memcached: An Experimental Performance Analysis. 2014 IEEE International

Conference on Communications (ICC 2014), 1154–1159. https://doi.org/10.1109/ICC.2014.6883477

Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., and Xie, Y. (2016). PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. Proceedings of the 2016 43rd International Symposium on Computer Architecture (ISCA 2016), 27–39. https://doi.org/10.1109/ISCA.2016.13

Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Noh, S. H., Sang Lyul Min, Yookun Cho, and Chong Sang Kim. (2001). LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers, 50(12), 1352–1361. https://doi.org/10.1109/TC.2001.970573

Fattahi, S., Yazdani, R., and Vahidipour, S. M. (2019). Discovery of Society Structure in a Social Network Using Distributed Cache Memory. Proceedings of the 2019 5th International Conference on Web Research (ICWR 2019), 264–269. https://doi.org/10.1109/ICWR.2019.8765289

Fitzpatrick, B. (2004). Distributed Caching with Memcached. Linux Journal, 2004(124), 5.

Hennessy, J. L., and Patterson, D. A. (1998). Computer Organization & Design, The Hardware/Software Interface. Morgan Kaufmann Publishers.

Hoff, T. (2009). Facebook's Memcached Multiget Hole: More Machines != More Capacity. High Scalability. http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html

Jinhyuk Yoon, Sang Lyul Min, and Yookun Cho. (2002). Buffer Cache Management: Predicting the Future from the Past. Proceedings International Symposium on Parallel Architectures, Algorithms and Networks 2002 (I-SPAN'02), 105–110. https://doi.org/10.1109/ISPAN.2002.1004268

Johns, M. (2015). Getting Started with Hazelcast (2nd ed.). Packt Publishing.

Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-Ur-Rahman, M., Islam, N. S., Ouyang, X., Wang, H., Sur, S., and Panda, D. K. (2011). Memcached Design on High Performance RDMA Capable Interconnects. Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11), 743–752. https://doi.org/10.1109/ICPP.2011.37

Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. (1999). On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99), 134–143. https://doi.org/10.1145/301453.301487

Mark Mulcahy. (2017). Big Data - Interesting Statistics, Facts & Figures. https://www.waterfordtechnologies.com/big-data-interesting-facts/

Martins, A., Penny, W., Weber, M., Palomino, D., Mattos, J., Porto, M., Agostini, L., and Zatt, B. (2017). Cache Memory Energy Efficiency Exploration for the HEVC Motion Estimation. Proceedings of the 2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC). https://doi.org/10.1109/SBESC.2017.11

Murugesan, S. (2007). Understanding Web 2.0. IT Professional, 9(4), 34–41. https://doi.org/10.1109/MITP.2007.78

Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., and Kim, H. (2017). GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. Proceedings of the 2017 International Symposium on High-Performance Computer Architecture (HPCA 2017), 457–468. https://doi.org/10.1109/HPCA.2017.54

Panda, P., Patil, G., and Raveendran, B. (2016). A Survey on Replacement Strategies in Cache Memory for Embedded Systems. Proceedings of the 2016 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER 2016), 12–17. https://doi.org/10.1109/DISCOVER.2016.7806218

Petrovič, J. (2008). Using Memcached for Data Distribution in Industrial Environment. 3rd International Conference on Systems (ICONS 2008), 368–372. https://doi.org/10.1109/ICONS.2008.51

Sedlacek, J., and Hurka, T. (2020). VisualVM. http://visualvm.github.io

Swain, D., Marar, S., Motwani, N., Hiwarkar, V., and Valakunde, N. D. (2018). CWRP: An Efficient and Classical Weight Ranking Policy for Enhancing Cache Performance. Proceedings of the 2017 4th International Conference on Image Information Processing (ICIIP 2017), 394–399. https://doi.org/10.1109/ICIIP.2017.8313747

URL-1, The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 107. (2017). Java Community Process. https://www.jcp.org/en/jsr/detail?id=107

URL-2, Ehcache. (2020). http://www.ehcache.org

URL-3, Guava: Google Core Libraries for Java. (2020). Google. https://github.com/google/guava

URL-4, Benchmarks. (2020). Cache2k. https://cache2k.org/benchmarks.html

URL-5, Open Source In-Memory Computing Platform - Apache Ignite. (2020). Apache. https://ignite.apache.org

URL-6, May 2009 Stack Exchange Data Dump. (2009). Stack Exchange. https://archive.org/details/stackexchange-ea45080eab61ab465f647e6366f775bf25f69a61

Wang, Z., McKinley, K. S., Rosenberg, A. L., and Weems, C. C. (2002). Using the Compiler to Improve Cache Replacement Decisions. Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02), 199.

Wong, W. A., and Baer, J.-L. (2000). Modified LRU Policies for Improving Second-Level Cache Behavior. Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), 49–60. https://doi.org/10.1109/HPCA.2000.824338

Yeung, K. H., and Ng, K. W. (1997). An Optimal Cache Replacement Algorithm for Internet Systems. Proceedings of 22nd Annual Conference on Local Computer Networks, 189–194. https://doi.org/10.1109/LCN.1997.630987