JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 16, No. 4, December 2022, pp. 194-198

Utilizing Temporal Locality for Hash Tables with Circular Chaining

Changwoo Pyo*
Department of Computer Engineering, Hongik University, Seoul, Korea
pyo@hongik.ac.kr

Taehwan Kim
Ultrasound Division, Siemens Healthineers Ltd., Koreal, Korea
taehwan.kim@siemens-healthineers.com

Abstract
A hash table with separate chaining typically adopts linearly linked lists as buckets to resolve collisions. This study demonstrates that converting bucket structures from linear to circular chaining enables buckets to utilize temporal locality and improve search performance. Unlike linear chaining, circular chaining can track the most recently accessed entry and preserve the reachability of all bucket entries without complicated data structures and operations. We defined temporal locality interval (TLI) to represent the period during which subsequent bucket access repeats itself on a single entry. We analyzed the average search cost using the TLI length and load factor. The average search cost converges to the minimum when the TLI length dominates the load factor. In our experiments using the SPEC CPU 2006 benchmark suite, circular chaining manifested 1.14 comparisons, reducing the cost of linear chaining by 45.71% when the load factor was 0.99. The improvement is notable, particularly for tables with a high load factor and uneven distribution of bucket sizes.

Category: Algorithms and Complexity

Keywords: Bucket structure; Circular chaining; Hash table; Linear chaining; Locality; Temporal locality interval

I. INTRODUCTION

A hash table with separate chaining typically adopts linearly linked lists as buckets to resolve collisions [1]. When a bucket has more than one entry, the table relies on sequential probing to search for a specific entry. If the bucket sizes are not significantly different, the average search cost is proportional to the average bucket size, which is also known as the load factor. Programmers create hash tables with sufficient buckets to maintain low load factors. Java's hash map class rehashes table entries after doubling the table size when the load factor exceeds 0.75 [2].

Researchers have proposed substitute bucket structures for linear chaining to reduce search costs. For instance, buckets implemented as string pools of linear arrays take advantage of spatial locality in cache memory [3]. In Java 8 and later, the hash map converts large-linked list buckets to red-black balanced trees [4]. This structural change limits the search cost to tree height but requires additional operations for balancing. Self-organizing linear chaining buckets [5] move the most recently accessed record to the front or swap it with its predecessor to utilize temporal locality [6]. Modifying at least three reference variables is necessary for the moving-to-head and swapping operations. Lists of lists

Open Access http://dx.doi.org/10.5626/JCSE.2022.16.4.194 http://jcse.kiise.org

Received 18 July 2022; **Accepted** 06 December 2022
*Corresponding Author

pISSN: 1976-4677 eISSN: 2093-8020

(LOLs) [7] stratify the self-organizing list into sublists to further utilize temporal locality. LOLs move an accessed entry and the sublist containing it by altering at least six pointer variables. In general, elaborate bucket structures require more operations than linear chaining to maintain their properties. As a side effect, search performance may deteriorate instead of improving in the absence of sufficient re-accesses.

This study demonstrates that converting the bucket structure from linear to circular chaining enables buckets to utilize temporal locality and improve search performance. We focus on search cost because search is the most frequently used table operation on its own and precedes insertion and deletion. After analyzing the average search cost, we present the experimental results and discuss their implications.

II. UTILIZATION OF TEMPORAL LOCALITY

A bucket must track its most recently accessed entry to revisit it immediately at the lowest possible cost. Simultaneously, each bucket entry should be reachable from the bucket header. Circular chaining can satisfy these two requirements without the need for complicated data structures and operations; the bucket header is maintained to reference the entry accessed last, and the circularity of the bucket structure preserves the reachability. If the subsequent bucket search does not target the entry, probing the bucket begins from the entry that the header references. The search can scan the entire bucket from wherever it begins. If a table does not have to differentiate the beginning from the last visited entry, there is no change in the necessary memory for bucket headers compared to linear chaining. Otherwise, a bucket header has to keep two separate references—its initial entry and the most recently visited entry; consequently, the storage requirement increases proportionally to the number of bucket headers. However, note that the number is constant no matter how many entries a table or bucket has.

III. ANALYSIS OF AVERAGE SEARCH COST

Let the access sequence be a time-series data tagged by an insertion, search, or deletion operation. Access sequences are observable from the table's or bucket's point of view. We define temporal locality interval (TLI) as a maximal continuous segment of the same data irrespective of tags in a bucket's access sequence. A delete-tagged datum terminates a TLI, although reinsertion of the deleted datum follows immediately. Reinsertion after deletion initiates a new TLI. A TLI represents the period during which subsequent bucket access repeats itself on a single entry. Because the first access in a TLI amortizes [8] the expenses for the remaining accesses of the TLI, we drew the average search cost from the total cost of the TLI. The cost metric is the number of table entries compared to the data of an access sequence.

We assume that table entries are distributed uniformly among buckets and are equally likely to be accessed. Additionally, suppose that the bucket headers reference arbitrary bucket entries. The first search for an entry in a TLI with circular chaining is not different from that with linear chaining. Let the cost be $E_L(\alpha)$, where α denotes the load factor. Then, the entry becomes immediately accessible by one comparison because the first access in the TLI has updated the bucket header to reference the entry. $E_C(\alpha, K)$, the expected cost of searching for an entry with circular chaining is $(E_L(\alpha) + K - 1)/K$, where K is the average length of TLIs. $E_L(\alpha)$ is approximately $1 + \alpha/2$, as derived in [1]. Therefore,

$$E_C(\alpha, K) \approx 1 + \frac{\alpha}{2K}.$$

The cost converges to the minimum when TLI length K is sufficiently larger than load factor α. For instance, if every bucket has at most one entry, K is equal to the number of all accesses to the bucket entry of an access sequence. While the entry remains in the table, K is virtually infinite, and the cost is asymptotically equal to one. For $K = 1$, circular chaining has neither an advantage nor disadvantage over linear chaining.

IV. EXPERIMENTS

We prepared test data with ample temporal locality from branch vectors observable at runtime. A branch vector consists of a branch instruction's source and destination addresses. Programs for architectural simulations [9] and software security [10] frequently monitor branch instructions by comparing branch vectors with others previously confirmed to exist or be safe. A branch instruction in a loop repeats as many times as the number of iterations. Thus, the monitoring numerously observes identical vectors. We executed SPEC CPU 2006 benchmark programs instrumented using LLVM [11] and collected the branch vectors of indirect jump instructions as access sequences.

In the experiment, we constructed a hash table by inserting branch vectors according to the order of their appearance in an access sequence. Every bucket header stored the references to the bucket entry inserted or accessed last. Branch vectors served as keys, and no two table entries shared identical vectors. The tested tables had 2^m, m = 5, …, 10, buckets. The hash function h_m corresponding to a table of size 2^m took the lower-order

m bits from the exclusive-OR of the two components of a branch vector. The hash functions, made of two logical instructions, were fast but did not produce uniformly distributed hash addresses for some benchmark programs.

During the table construction, we counted the visited table entries for each element of the access sequence. After building a table, we tested it using the same access sequence used for table construction. As all vectors were already in the table, the results exhibit the costs of successful searches. Furthermore, we measured bucket sizes and TLI lengths. We repeated the table construction and lookup testing twice: once for linear chaining and again for circular chaining.

V. RESULTS AND DISCUSSION

Table 1 presents the experimental results for six table sizes. The second and third columns show the load factors geometrically averaged over 20 benchmarks and the numbers of programs with statistically nonuniform bucket sizes, respectively [12]. The costs of table construction are listed in the following three columns, including the reduction rate achieved by changing the bucket structure from linear to circular. The rest of the table is the same as the preceding three columns in nature but was measured in lookup-only tests.

Circular chaining cost less than linear chaining for all the table sizes. It manifested 1.14 comparisons for a search near the minimum when the average load factor was 0.99, as in the case of 512 buckets. In contrast, linear chaining needed 2.10 comparisons. Circular chaining reduced the cost by 45.71%, and similarly, it reduced the cost of table construction by 41.33% for tables with 512 buckets.

The installation of more buckets in the tables reduced the load factor. At the same time, the possibility of collision also decreased, and the TLIs were consequently extended. Fig. 1 shows that the TLIs grew as table size increased. Circular chaining took advantage of both prolonged TLIs and reduced bucket sizes, whereas linear chaining, which is incapable of using TLIs, could benefit only from shortened buckets. Therefore, circular chaining approached the minimum cost faster than linear chaining as the load factor decreased, as shown in Fig. 2(a). In addition, Fig. 2(b) shows that search costs dropped rapidly to the minimum when TLIs grew longer.

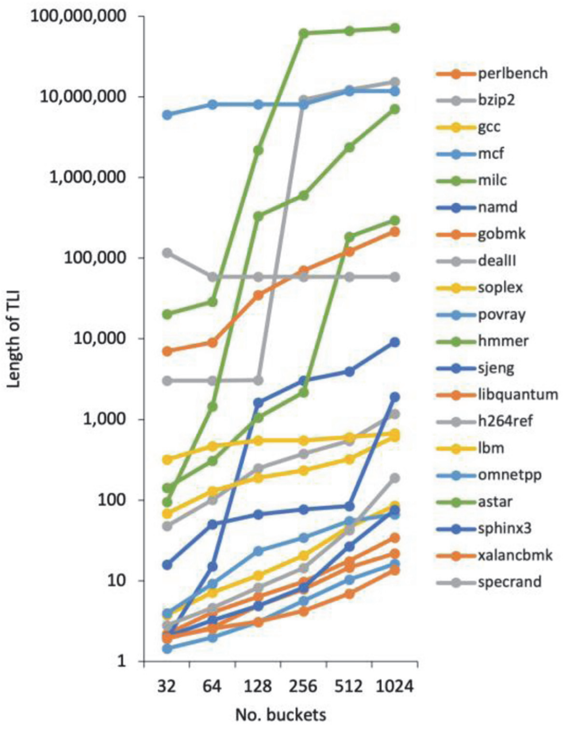
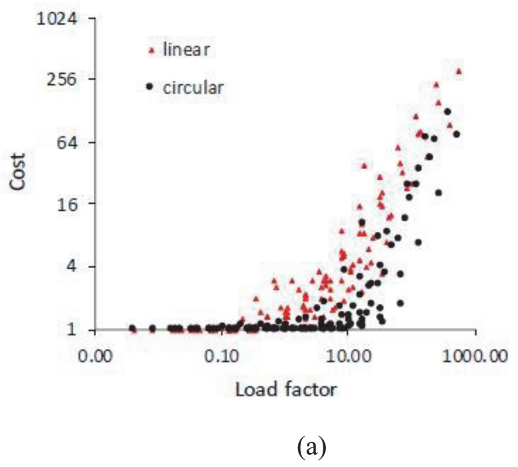


Fig. 1. Temporal locality interval (TLI) lengths by table sizes (number of buckets).

Table 1. Results of experiments

No. of buckets	Load factor	No. of skewed programs[a]	Construction cost			Lookup-only cost		
			Circular	Linear	Reduction[b] (%)	Circular	Linear	Reduction[b] (%)
32	15.84	9	4.11	6.92	40.61	4.26	9.56	55.44
64	7.92	9	2.56	4.46	42.60	2.62	5.60	53.21
128	3.96	7	1.96	3.52	44.32	1.81	3.90	53.59
256	1.98	5	1.42	2.59	45.17	1.39	2.76	49.64
512	0.99	4	1.15	1.96	41.33	1.14	2.10	45.71
1024	0.49	3	1.06	1.53	30.72	1.06	1.63	34.97

The cost metric is the number of comparisons between a searched datum and table entries.
[a]Chi-square test [12] determined the uniformity of the bucket size distribution of 20 benchmark programs.
[b]Reduction is percentile cost-down by circular chaining for linear chaining.



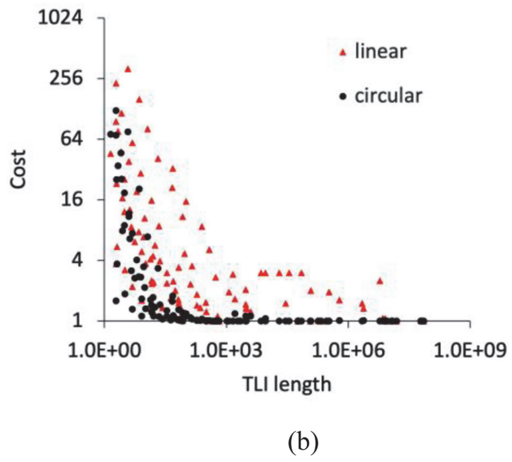


Fig. 2. Access costs by (a) load factors and (b) TLI lengths without identifying benchmarks and table sizes.

Circular chaining did not show noticeable differences between the table building and lookup-only tests because both can use TLIs. By contrast, the costs of table construction and searches, shown in the fifth and eighth columns of Table 1, were distinct for linear chaining. The insertion of an entry opened a new TLI during table construction and updated the bucket header to reference the entry. Thus, linear chaining could also utilize the TLI beginning with insertion during table construction, although its utilization is limited compared with circular chaining. However, searches with linear chaining could not benefit from a TLI as much as table construction because the bucket header did not track the most recently accessed entry during lookup-only tests.

Furthermore, circular chaining is superior to linear chaining in terms of space efficiency and immunity to nonuniform bucket sizes. The experimental results show that the performance of circular chaining is comparable to or better than that of four-times larger tables with linear chaining. Note also that smaller tables had more agglomerated buckets than larger tables. It might take longer for large buckets to reach an entry at the beginning of a TLI, but revisiting the entry in the remaining TLI entailed minimum costs regardless of bucket size because a bucket header always referenced the most recently accessed entry. Circular chaining can hide the performance degradation caused by significantly larger buckets if immediate re-accesses are available. For dynamic resizing tables, circular chaining can delay rehashing longer than linear chaining can.

VI. CONCLUSION

The experimental results indicate that increasing the number of buckets expands temporal locality, which circular chaining harnessed to improve performance. The improvement was notable, particularly for tables with a high load factor and uneven bucket size distribution, providing a trade-off between the uniformity and speed of hash functions for overall performance. We expect that the temporal locality at the bucket level also brings locality into cache memory, and thus circular chaining boosts cache memory utilization.

ACKNOWLEDGEMENTS

This research was supported by 2021 Hongik University Research Fund and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2019R1F1A105701312).

REFERENCES

1. E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*, 2nd ed. Summit, NJ: Silicon Press, 2007, pp. 458-488.
2. Oracle, “HashMap (Java Platform SE 8),” 2022 [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.
3. N. Askitis and J. Zobel, “Cache-conscious collision resolution in string hash tables,” in *String Processing and Information Retrieval*. Heidelberg, Germany: Springer, 2005, pp. 91-102
4. katleman, “OpenJDK src/share/classes/java/util/HashMap.java," 2014 [Online]. Available: https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java/#l1791.
5. J. McCabe, “On serial files with relocatable records,” *Operations Research*, vol. 13, no. 4, pp. 609-618, 1965.
6. P. J. Denning, “Working set analytics,” *ACM Computing*

Surveys, vol. 53, no. 6, pp. 1-36, 2021.
7. A. Amer and B. J. Oommen, "Lists on lists: a framework for self-organizing lists in environments with locality of reference," in *Experimental Algorithms*. Heidelberg, Germany: Springer, 2006, pp. 109-120.
8. R. E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306-318, 1985.
9. R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: a survey," *ACM Computing Surveys*, vol. 29, no. 2, pp. 128-170, 1997.
10. N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: precision, security, and performance," *ACM Computing Surveys*, vol. 50, no. 1, pp. 1-33, 2017.
11. C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of International Symposium on Code Generation and Optimization*, San Jose, CA, 2004, pp. 75-86.
12. National Institute of Standards and Technology, "NIST/SEMATECH e-Handbook of Statistical Methods," 2012 [Online]. Available: https://www.itl.nist.gov/div898/handbook/.

Changwoo Pyo

Changwoo Pyo received Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, IL, USA, in 1989. He was a research fellow in the US Army Corps of Engineers from 1989 until 1991. Since then, He has been a professor at the Department of Computer Engineering, Hongik University, Seoul, Korea. His research interests include program analysis, transformation, and compilation for higher performance and autonomous program protection for software security.

Taehwan Kim

Taehwan Kim received Ph.D. in computer engineering from Hongik University, Seoul, Korea, in 2019. He continued his work as a post-graduate research fellow at the Hongik University Research Institute of Science and Technology before he joined Siemens in July 2019. Since then, he has served as a cybersecurity officer and data privacy expert in the Ultrasound Global R&D Manufacturing Center SUSKO, Siemens Healthineers. His research interests include secure software design and secure coding, data security and privacy, and program analysis and optimization.