Vol.2016-HPC-155 No.35
2016/8/10

Performance assessment of highly concurrent sorted linked list with good spatial locality (Unrefereed Workshop Manuscript)

Mohamed Amin Jabri[1] Osamu Tatebe[2]

Abstract: In this paper, we present a highly concurrent sorted linked list using a lock-free approach. To achieve a good spatial locality in our design, the list is arranged into linked chunks of memory with pre-allocated fixed number of entry slots. Entries within a chunk are kept logically sorted and an inter-chunk procedure keeps them from becoming too sparse or too dense. The performance of our design under stress workloads is also presented.

1. Introduction

In this work, we present a highly concurrent, lock-free based sorted linked list optimized for cache spatial locality, by splitting the list into a linked contiguous chunks of memory holding logically sorted sub-list. A higher-level representation of such linked list is depicted in figure 1. This figures shows three linked chunks of memory each holding a contiguous logical sub-list for increased spatial locality. Initially, the list contained two linked chunks of memory, but due to a balancing operation (chunk split in this case), the dense chunk was replaced with two chunks.

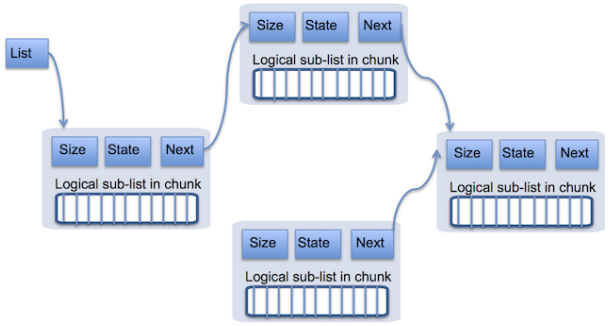

Fig. 1 Architecture of a linked list with improved spatial locality arranged as linked sub-lists of logically sorted lists allocated over fixed sized contiguous chunks of memory.

The benchmarked scheme in this paper, is based on the work of T. Harris [1], which present a simple yet a fast lock-free concurrent linked-list implementation with a linearizable data insertion and deletion. The latter operation is done in two-steps. First, the entrys next field containing a pointer to the next entry is marked. Second, the marked entry is disconnected from the list. All operations, including insertion, deletion and retrieval, relies on a Search procedure which given a search key returns a left entry holding the greatest key which less than the search key and a right entry holding the search key or the smallest key greater than the search key. The Search procedure unlinks from the list all the logically deleted but not yet unlinked entries from the list and ensures that the returned right entry is the immediate entry after the returned left entries.

For chunk management and inter-chunks balancing operation, the benchmarked scheme in this paper is based on the work of A. Braginsky [2, 3], but instead of using hazard pointers for memory management and reclamation (for instance when recycling deleted entries), we avoid it altogether until chunks are disconnected from the list in a first implementation and with automatic entry reference counting in a second implementation.

2. Evaluation

In this section, we provide a performance evaluation of our proposed lock-free sorted linked list under a stress micro-benchmark. The stress micro-benchmark used in our evaluation is based on Google benchmark [4], a micro-benchmarking support library which lets developers write time-based (as opposed to iterations-based) benchmarking stress test in a similar fashion to unit-testing. In a time-based benchmark, the number of iterations invoking a given test method is dynamically computed at run-time and ensures that the benchmarked test-case or function call keeps running such that its Wall time is not below a specified minimum time value (*benchmark_min_time*) or its CPU time is at least five times the value of *benchmark_min_time*. In our experiments, the minimum time *benchmark_min_time* is set to a *one millisecond*.

We conducted our benchmarks on a Linux server with the characteristics shown in table 1. Also, we constrained all benchmark threads to a one *NUMA* domain using the *numactl* Linux command. For each benchmark (*Put*, *Get*, *Delete* and *Mixed-load*), the performance results for both the lock-free linked list with and without the Automatic Reference Counting are measured, in order to assess the impact of reference counting as a memory man-

[1] Center for Computational Sciences, University of Tsukuba
[2] Faculty of Engineering, Information and Systems, University of Tsukuba

agement strategy.

Operating System	Centos 7 (kernel 3.10.0)
Compiler	llvm 3.8.0 (std=c++14)
CPU	Intel Xeon E5-2665 (2 sockets, 16 cores, 2.39 Ghz)
Memory	64 GB

Table 1 Benchmarking server environment.

Figure 2 depicts the number of *Put* operations per seconds for different concurrent thread counts (ranging from 1 to 32 threads) and data sizes, which is the total number of key-value pairs (ranging from 16 to 4096) to be inserted in the initially empty list. Data is distributed evenly among all the spawned threads. The aggregated wall time and number of *Put* operations executed by the benchmark are reported, in order to compute the number of operations per seconds shown in figure 2.

Figure 3 depicts the number of *Get* operations per seconds for different concurrent thread counts (ranging from 1 to 32 threads) and data sizes ranging from 16 to 4096 key-value pairs retrieved from a prepopulated list containing 4096 entries. The data retrieval workload is distributed evenly among all the spawned threads. The aggregated wall time and number of *Get* operations executed by the benchmark are reported, in order to compute the number of operations per seconds shown in figure 3.

The performance overhead of the *Delete* operation in our benchmark in terms of deleted entries per unit of time with respect to different configurations: number of concurrent thread (ranging from 1 to 32 threads) and number of deleted data (ranging from 16 to 4096 entries), from an initially prepopulated list containing 4096 entries, is shown in figure 4. The deletion workload is distributed evenly among all the spawned threads. The aggregated wall time and number of *Delete* operations executed by the benchmark are reported, in order to compute the number of operations per seconds depicted in figure 4.

Figure 5 shows the behavior of our lock-free sorted linked list, in terms od operations per unit of time with respect to different settings of concurrent thread count ranging from 1 to 32 threads and data sizes (from 16 to 4096 key-value pair entries), under a mixed workload composed of: 60 percent of *Put* operations, 30 percent of *Get* operations and 10 percent of *Delete* operations. Initially, the list was prepopulated with 1024 key-value pairs. The reported operations per unit of time correspond to the aggregated performance across all the spawned threads.

In all our benchmarks (*Put*, *Get* and *Delete* operations), only the non-reference-counted version of our lock-free linked list scales well with the number of threads even with oversubscription (case with 16 and 32 threads). For the mixed-load benchmark, multithreading introduce a small performance increase compared to the single threaded version of the automatic reference counting based version of our lock-free linked list. Also, for all benchmarks we see a performance decrease when the data size increases for both versions of our lock-free linked list.

Our stress benchmarks shows that reference counting as a strategy for memory management and reclamation, in our implementation, hugely impacts performance compared to the non-reference-counted version. Also, our non-reference-counted lock-free linked list exhibits a good performance (millions of operations per unit of time with small data sizes) and scales well with threading.

3. Conclusion

In this paper, we presented a highly concurrent lock-free sorted linked list based on the work of T. Harris [1] and A. Braginsky [2,3], which leverage data spatial locality. This is achieved by arranging the linked list into linked sub-lists of contiguous fixed sized chunks of memory. Each sub-list is logically sorted and its size (number of entries) is kept within a minimum and maximum thresholds (using an inter-chunks balancing operations). Also, in our implementation we consider memory reclamation and recycling using systematic entries reference counting. The performance evaluation of our implementation under a stress benchmark shows a good scalability with thread count increase for the non-reference counted version of our implementation. Additionally, our results show how memory reclamation and management through automatic reference counting degrades considerably performance.

References

[1] Harris, T. L.: A Pragmatic Implementation of Non-blocking Linked-Lists, *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, London, UK, UK, Springer-Verlag, pp. 300–314 (online), available from ⟨http://dl.acm.org/citation.cfm?id=645958.676105⟩ (2001).

[2] Braginsky, A. and Petrank, E.: Locality-conscious Lock-free Linked Lists, *Proceedings of the 12th International Conference on Distributed Computing and Networking*, ICDCN'11, Berlin, Heidelberg, Springer-Verlag, pp. 107–118 (2011).

[3] Braginsky, A. and Petrank, E.: Locality-conscious Lock-free Linked Lists, Online full paper version: `http://www.cs.technion.ac.il/~erez/Papers/lf-linked-list-full.pdf` (2010).

[4] Benchmark: A microbenchmark support library, Google (online), available from ⟨https://github.com/google/benchmark⟩ (accessed 2016-7-6).

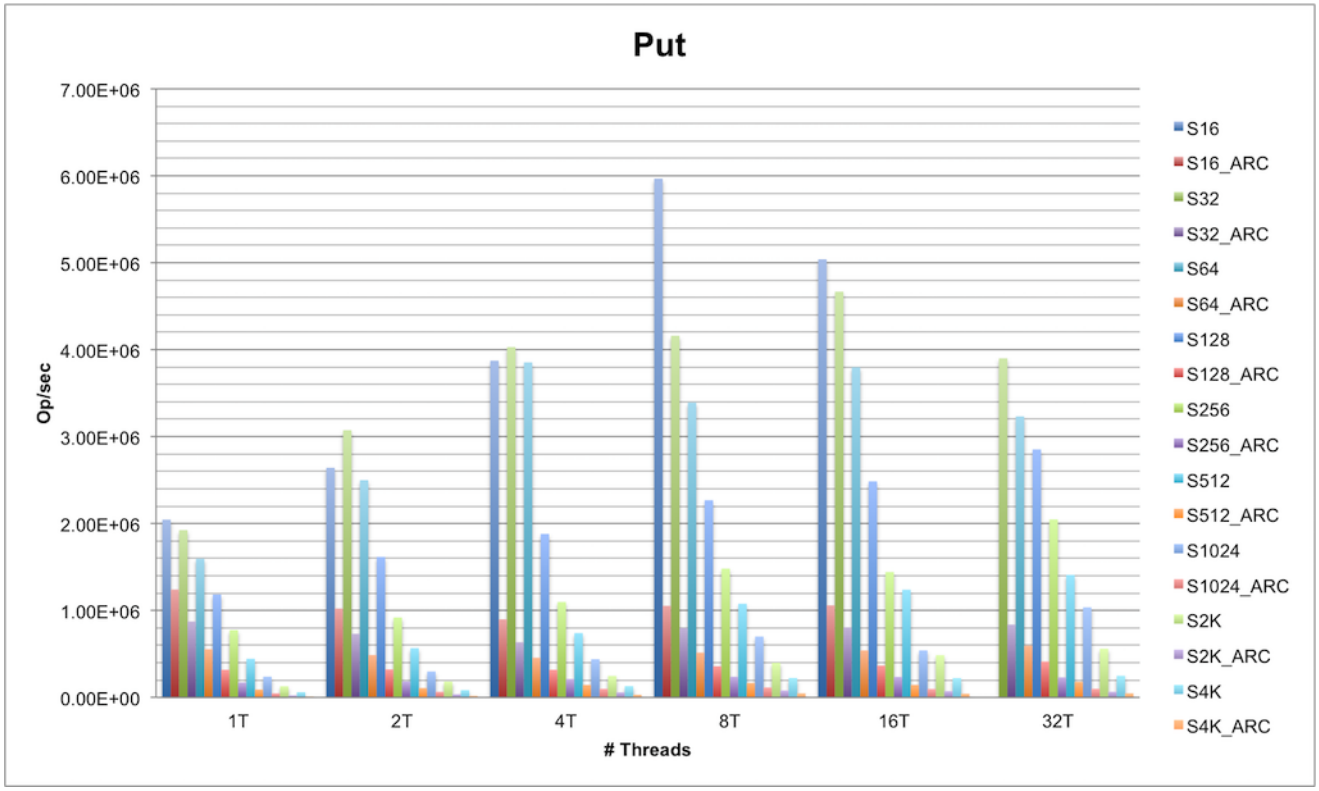

Fig. 2 Put operations under stress test with different configurations of concurrent threads number and data size (number of key-value pairs). Shown Operation per seconds are the aggregated measurement across all threads for both the lock-free linked list with and without Automatic Reference Counting (ARC suffix).

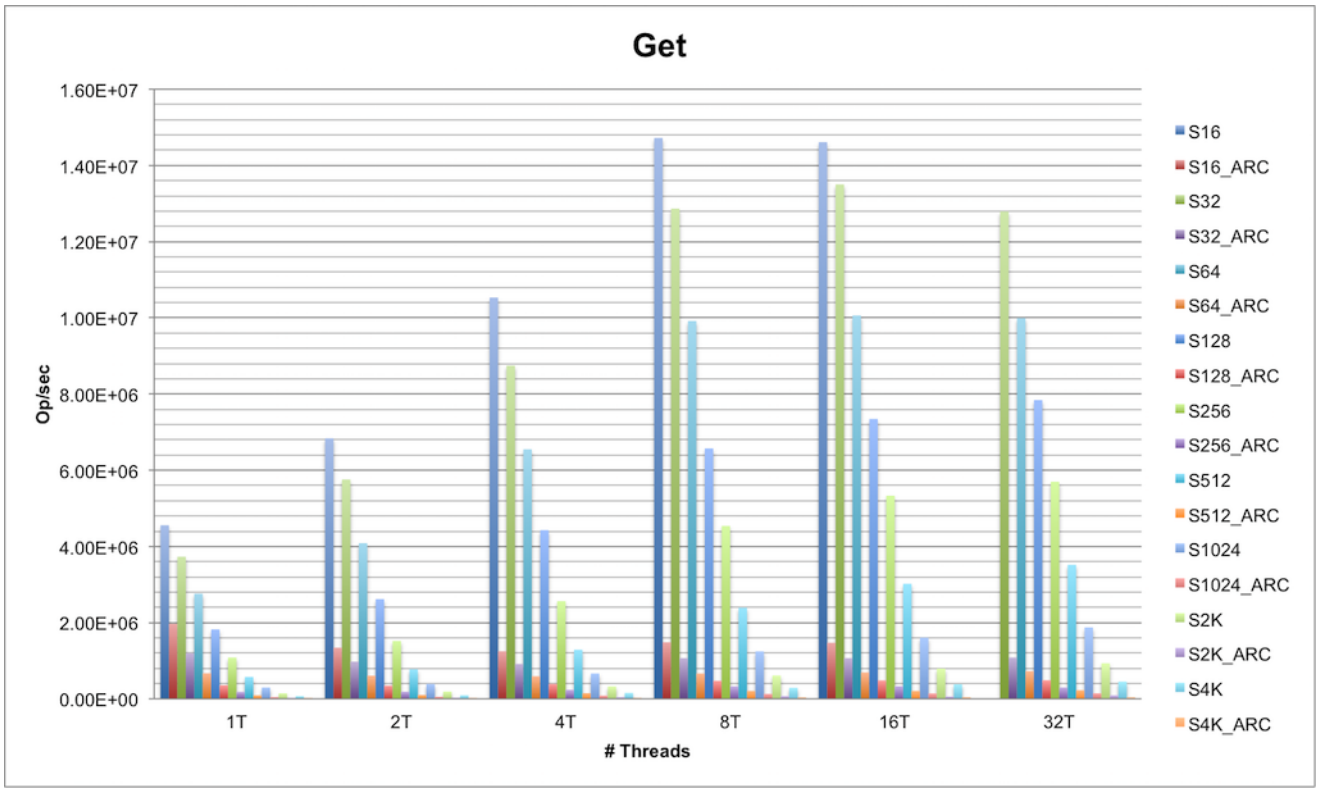

Fig. 3 Get operation under stress test with different configurations of concurrent threads number and data size (number of key-value pairs). Shown operation per seconds are the aggregated measurement across all threads for both the lock-free linked list with and without automatic reference counting (ARC suffix).

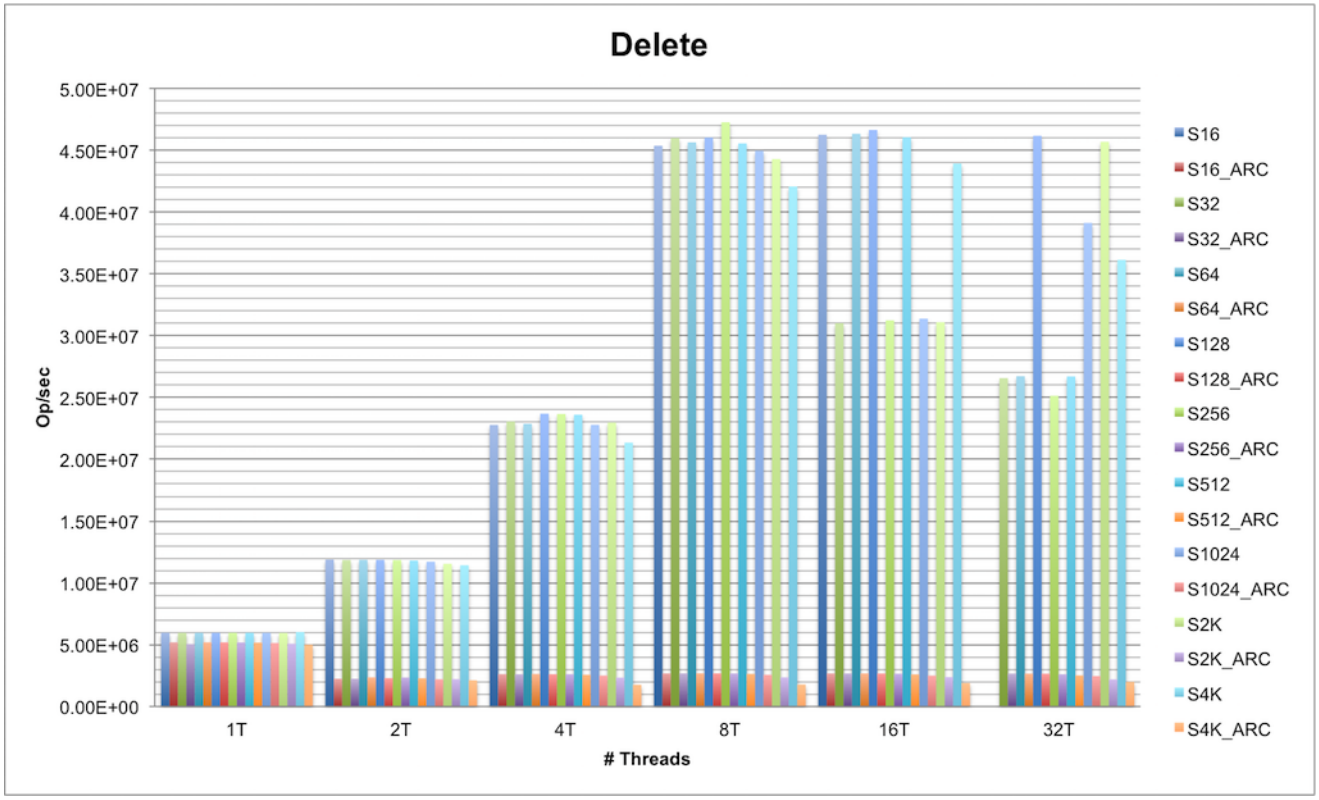

Fig. 4 Delete operation under stress test with different configurations of concurrent threads number and data size (number of key-value pairs). Shown operation per seconds are the aggregated measurement across all threads for both the lock-free linked list with and without automatic reference counting (ARC suffix).

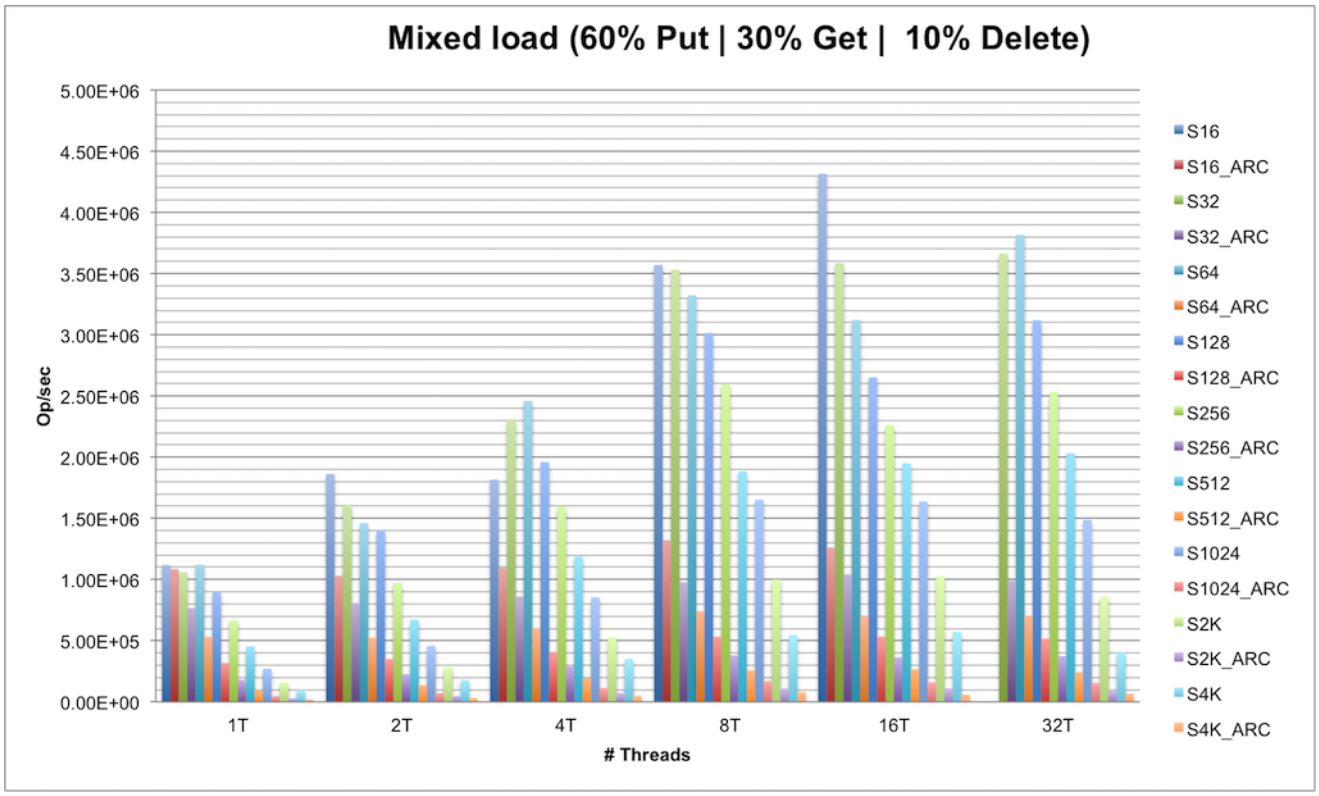

Fig. 5 Mixed operations set (Put 60%/Get 30%/Delete 10%) under stress test with different configurations of concurrent threads number and data size (number of key-value pairs). Shown operation per seconds are the aggregated measurement across all threads for both the lock-free linked list with and without automatic reference counting (ARC suffix).