A Distributed Hash Table for Shared Memory

Wytse Oortwijn$^{(\boxtimes)}$, Tom van Dijk, and Jaco van de Pol

Formal Methods and Tools, Department of EEMCS, University of Twente,
P.O.-box 217, 7500 AE Enschede, The Netherlands
{w.h.m.oortwijn,t.vandijk,j.c.vandepol}@utwente.nl

Abstract. Distributed algorithms for graph searching require a high-performance CPU-efficient hash table that supports `find-or-put`. This operation either inserts data or indicates that it has already been added before. This paper focuses on the design and evaluation of such a hash table, targeting supercomputers. The latency of `find-or-put` is minimized by using one-sided RDMA operations. These operations are overlapped as much as possible to reduce waiting times for roundtrips. In contrast to existing work, we use linear probing and argue that this requires less roundtrips. The hash table is implemented in UPC. A peak-throughput of 114.9 million op/s is reached on an Infiniband cluster. With a load-factor of 0.9, `find-or-put` can be performed in 4.5 μs on average. The hash table performance remains very high, even under high loads.

Keywords: Distributed hash table · High-performance computing · Partitioned global address space · Remote direct memory access

1 Introduction

A *hash table* is a popular data structure for storing maps and sets, since storing and retrieving data can be done with amortised time complexity $\mathcal{O}(1)$ [2]. A *distributed hash table* is a hash table that is distributed over a number of workstations, connected via a high-performance network. This has the advantage that more memory is available, at the cost of slower accesses due to network latency and bandwidth limitations. In High Performance Computing (HPC) it is desirable to have a fast and scalable distributed hash table, as it enables many distributed algorithms to be implemented efficiently.

Nowadays high-performance networking hardware like Infiniband [7] is available. Infiniband supports Remote Direct Memory Access (RDMA), which allows computers to directly access the memory of other machines without invoking their CPUs. Moreover, RDMA supports zero-copy networking, meaning that no memcopies are performed [14]. Experimental results show that one-sided RDMA is an order of magnitude faster compared to standard Ethernet hardware [10]. Furthermore, scaling along high-performance Infiniband hardware is comparable in price to scaling along standard Ethernet hardware [10]. In this paper, we target supercomputers, i.e. many-core machines connected via Infiniband.

The Partitioned Global Address Space (PGAS) programming model combines the shared and distributed memory models. Each process hosts a local

block of memory. The PGAS abstraction combines all local memory blocks into a single global address space, thereby providing a global view on the memory. PGAS can make use of RDMA if used in a distributed setting [5]. In that case, machine-local accesses to the global address space are handled via standard memory operations, and remote accesses are handled via *one-sided* RDMA. Several PGAS implementations provide support for RDMA, including OpenSHMEM [1] and UPC [4]. We use UPC, since it supports asynchronous memory operations.

Our goal is to implement a distributed hash table for the PGAS abstraction that supports a single operation, namely `find-or-put`, that either inserts data when it has not been inserted already or indicates that the data has been added before. If necessary, `find-or-put` could easily be split into two operations `find` and `insert`. Furthermore, the hash table should require minimal memory overhead, should be CPU-efficient, and `find-or-put` should have minimal latency.

Our motivation for designing such a hash table is its use in distributed symbolic verification (e.g. model checking), which only requires a `find-or-put` operation and garbage collection in a stop-the-world scenario. Garbage collection is, however, omitted in the design of `find-or-put` presented in this paper. We tried to minimize the number of roundtrips required by `find-or-put` while keeping the hash table CPU-efficient by not relying on memory polling. Many existing implementations are, however, either CPU-intensive [9] or require more roundtrips [10,15], which motivated this research. We use linear probing and argue that this scheme requires less roundtrips compared to alternative hashing schemes. Furthermore, the design of `find-or-put` is more widely applicable to any sort of memory-intensive application requiring a hash table, of which distributed model checking is merely an example.

Previous work includes Pilaf [10], a key-value store that employs RDMA. Pilaf uses an optimised version of Cuckoo hashing to reduce the number of roundtrips. In Pilaf, lookups are performed via RDMA reads, but inserts are handled by the server. Nessie [15] is a hash table that uses Cuckoo hashing and RDMA both for lookups and inserts. HERD [9] is a key-value store that only uses one-sided RDMA writes and ignores the CPU bypassing features of RDMA to achieve higher throughput. FaRM [3] is a distributed computing platform that exposes the memory of all machines in a cluster as a shared address space. A hash table is built on top of FaRM that uses a variant of Hopscotch hashing.

This paper is organised as follows. Different hashing strategies are compared in Sect. 2 and we argue that linear probing requires the least number of roundtrips. Section 3 discusses the design of `find-or-put`. Section 4 shows the experimental evaluation of `find-or-put`, covering hash table efficiency with respect to latency, throughput, and the required number of roundtrips. Finally, our conclusions are summarised in Sect. 5.

2   Preliminaries

To achieve best performance, it is critical to minimize the number of RDMA roundtrips performed by `find-or-put` when targeting remote memory. This is

because the throughput of the hash table is limited to the throughput of the RDMA devices. Also, the waiting times for roundtrips contribute to the latency of `find-or-put`. In this section some notation is given, followed by a number of hashing strategies and their efficiencies with respect to the number of roundtrips.

2.1   Notation

A hash table $T = \langle b_0, \ldots, b_{n-1} \rangle$ consists of a sequence of buckets b_i usually implemented as a standard array. We denote the *load-factor* of T by $\alpha = \frac{m}{n}$, where m is the number of elements inserted in T and n the total number of buckets. A hash function $h : U \to R$ maps data from some universe $U = \{0,1\}^w$ to a range of keys $R = \{0, \ldots, r-1\}$. Hash tables use hash functions to map words $x \in U$ to buckets $b_{h(x)}$ by letting $r \leq n$. Let $x \in U$ be some word. Then we write $x \in b_i$ if bucket b_i contains x, and otherwise $x \notin b_i$. We write $x \in T$ if there is some $0 \leq i < n$ for which $x \in b_i$, and otherwise $x \notin T$. For some $x, y \in U$ with $x \neq y$ it may happen that $h(x) = h(y)$. This is called a *hash collision*. A hash function $h : U \to R$ is called a *universal hash function* if $\Pr[h(x) = h(y)] \leq \frac{1}{n}$ for every pair of words $x, y \in U$.

2.2   Hashing Strategies

Ideally only a single roundtrip is ever needed both for finding and inserting data. This can only be achieved when hash collisions do not occur, but in practice they occur frequently. HERD only needs one roundtrip for every operation [9], but at the cost of CPU efficiency, because every machine continuously polls for incoming requests. We aim to retain CPU efficiency to keep the hash table usable in combination with other high-performance distributed algorithms.

Chained hashing is a hashing scheme which implements buckets as linked lists. Insertions take $\mathcal{O}(1)$ time, but lookups may take $\Theta(m)$ in the worst case. It can be shown that lookups require $\Theta(1 + \alpha)$ time on average when a universal hash function is used [2]. Although constant, the average number of roundtrips required for an insert is thus more than one. Furthermore, maintaining linked lists brings memory overhead due to storing pointers.

Cuckoo hashing [11] is an open address hashing scheme that achieves constant lookup time and expected constant insertion time. Cuckoo uses $k \geq 2$ independent hashing functions $h_1, \ldots, h_k$ and maintains the invariant that, for every $x \in T$, it holds that $x \in b_{h_i(x)}$ for exactly one $1 \leq i \leq k$. Lookups thus require at most k roundtrips, but inserts may require more when all k buckets are occupied. In that case, a relocation scheme is applied, which may not only require many extra roundtrips, but also requires a locking mechanism, which is particularly expensive in a distributed environment. A variant on Cuckoo hashing, named *bucketized Cuckoo hashing* [13], enables buckets to contain multiple data elements, which linearly reduces the number of required roundtrips.

Hopscotch hashing [6] also has constant lookup time and expected constant insertion time. In Hopscotch every bucket belongs to a fixed-sized *neighbourhood*. Lookups only require a single roundtrip, since neighbourhoods are

consecutive blocks of memory. However, inserts may require more roundtrips when the neighbourhood is full. In that case, buckets are relocated, which may require many roundtrips and expensive locking mechanisms.

Linear probing examines a number of *consecutive* buckets when finding or inserting data. Multiple buckets, which we refer to as *chunks*, can thus be obtained with a single roundtrip. When there is a hash collision, linear probing continues its search for an empty bucket in the current chunk, and requests additional consecutive chunks if necessary. We expect chunks retrievals to require less roundtrips than applying a relocation scheme, like done in Hopscotch. Other probing schemes, like *quadratic probing* and *double hashing*, require more roundtrips, since they examine buckets that are nonconsecutive in memory.

Cache-line-aware linear probing is proposed by Laarman et al. [12] in the context of NUMA machines. Linear probing is performed on cache lines, which the authors call *walking-the-line*, followed by double hashing to improve data distribution. Van Dijk et al. [16] use a probe sequence similar to walking-the-line to implement `find-or-put`, used for multi-core symbolic verification.

3   Design and Implementation

In this section the hash table structure and the design of `find-or-put` are discussed. We expect linear probing to require less roundtrips than both Cuckoo hashing and Hopscotch hashing, due to the absence of expensive relocation mechanisms. We also expect that minimising the number of roundtrips is key to increased performance, since the throughput of the hash table is directly limited by the throughput of the RDMA devices. This motivates the use of linear probing in the implementation of `find-or-put`. Unlike [12], we only use linear probing, since it reduces latency compared to quadratic probing, at the cost of possible clustering. We did not observe serious clustering issues, but if clustering occurs, quadratic probing can still be used, at the cost of slightly higher latencies.

The latency of `find-or-put` depends on the waiting time for roundtrips to remote memory (which is also shown in Sect. 4). We aim to minimize the waiting times by overlapping roundtrips as much as possible, using asynchronous memory operations. Furthermore, the number of roundtrips required by `find-or-put` is linearly reduced by querying for *chunks* instead of individual buckets. We use constant values C to denote the *chunk size* and M to denote the maximum number of chunks that `find-or-put` takes into account. Figure 1 shows the design of `find-or-put`. Design considerations are given in the following sections.

3.1   Memory Layout

In our implementation, each bucket is 64 bits in size. The first bit is used as a flag to denote bucket occupation and the remaining 63 bits are used to store data. When inserting data, the occupation bit is set via a `cas` operation to prevent expensive locking mechanisms. If the hash table needs to support the storage of data elements larger than 63 bits, a separate shared data array could be used.

```
1 def find-or-put(data):
2   h ← hash(data)
3   s_0 ← query-chunk(0, h)
4   for i ← 0 to M − 1:
5     if i < M − 1
6       s_{i+1} ← query-chunk(i + 1, h)
7     sync(s_i)
8     for j ← 0 to C − 1:
9       if ¬occupied(p_{(i,j)})
10        addr ← (h + iC + j) mod kn
11        b ← new-bucket(data)
12        val ← cas(b_a, p_{(i,j)}, b)
13        if val = p_{(i,j)}
14          return inserted
15        elif data(val) = data
16          return found
17      elif data(p_{(i,j)}) = data
18        return found
19  return full
```

```
1 def query-chunk(i, h):
2   start ← (h + iC) mod kn
3   end ← (h + (i + 1)C − 1) mod kn
4   if end < start
5     return split(start, end)
6   else
7     S ← ⟨b_{start} ··· b_{end}⟩
8     P ← ⟨p_{(i,0)} ··· p_{(i,C−1)}⟩
9     return memget-async(S, P)
```

```
1 def split(start, end):
2   S_1 ← ⟨b_{start} ··· b_{kn−1}⟩
3   S_2 ← ⟨b_0 ··· b_{end}⟩
4   P_1 ← ⟨p_{(i,0)} ··· b_{(i,|S_1|−1)}⟩
5   P_2 ← ⟨p_{(i,|S_1|)} ··· p_{(i,C−1)}⟩
6   s_1 ← memget-async(S_1, P_1)
7   s_2 ← memget-async(S_2, P_2)
8   return ⟨s_1, s_2⟩
```

Fig. 1. The implementation of `find-or-put`, as well as the implementation of `query-chunk` and `split`, which are used by `find-or-put` to query on the ith chunk.

The data elements are then stored in the data array and the corresponding indices are indexed and stored in the hash table. In that case, an extra roundtrip is required by `find-or-put` to access the data array.

The atomic $\mathsf{cas}(\mathsf{B}, c, v)$ operation compares the content of a shared memory location B to a value c. If they match, v is written to B and the *former* value at location B is returned by `cas`. Otherwise, B is unchanged and its contents are returned. The $\mathsf{occupied}(b)$ operation simply checks if the occupation bit of a bucket b is set and the $\mathsf{new\text{-}bucket}(d)$ operation creates a new bucket with its occupation bit set to `true` and containing d as data.

Assuming that the hash table is used by n processes $t_1, \ldots, t_n$, we allocate a *shared* table $T = \langle b_0, \ldots, b_{kn-1} \rangle$ of buckets, such that each process owns k buckets. In addition, we allocate two-dimensional arrays $P_i = \langle p_{(0,0)}, \ldots, p_{(M-1,C-1)} \rangle$ on every process t_i in *private* memory, which we use as local buffers. The arrays P_i are furthermore cache line aligned. This minimizes the number of cache misses when iterating over P_i, thus reducing the number of data fetches from main-memory. Cache lines are typically 64 bytes in size, so 8 buckets fit on a single cache line. To optimally use cache line alignment we choose C to be a multiple of 8.

3.2 Querying for Chunks

In Fig. 1, when some process t_j queries a chunk, it transfers C buckets from the shared array T into P_j, so that t_j can examine the buckets locally. Because linear probing is used, several *consecutive* chunks might be requested.

The `query-chunk` operation is used to query the ith consecutive chunk and the `sync` operation is used to *synchronize* on the query, that is, waiting for its completion.

It may happen that $end < start$ (line 4 of `query-chunk`), in which case the chunk wraps around the kn-sized array T because of the modulo operator (line 2 and line 3). Then the query has to be split into two, as the chunk spans over two nonconsecutive blocks of shared memory. This is done by the `split` operation.

The `memget(S, P)` operation is supported by many PGAS implementations and transfers a block of shared memory S into a block of private memory P owned by the executing process. Then `memget-async` is a *non-blocking* version of `memget`, as it does not block further execution of the program while waiting for the roundtrip to finish. Instead, `memget-async` returns a handle that can be used by `sync`, which is a blocking operation used to synchronize on the roundtrip. This allows work to be performed in between calls to `memget-async` and `sync`. The `query-chunk` operation itself returns one or two handles, and `sync` can be used to synchronize on them.

3.3 Designing `find-or-put`

In Fig. 1, the call `find-or-put`(d) returns `found` when $d \in T$ before the call and returns `inserted` when $d \notin T$ before the call. Finally, `full` is returned when $d \notin T$ and d could not be inserted in any of the MC examined buckets.

The algorithm first requests the first chunk and, if needed, tries a maximum of $M-1$ more chunks before returning `full`. Before calling `sync`(s_i) on line 7, the next chunk is requested by calling `query-chunk`$(i + 1, \ d)$ on line 6. This causes the queries to overlap, which reduces the blocking times for synchronization on line 7 and thereby reduces the latency of `find-or-put`.

By iterating over a chunk, if a bucket $p_{(i,j)}$ is empty, `find-or-put` tries to write $data$ to the bucket b_a in shared memory via a `cas` operation (line 12). The *former* value of b_a is returned by `cas` (line 12), which is enough to check if `cas` succeeded (line 13). In this case, `inserted` is returned, otherwise the bucket has been occupied by another process in the time between the calls to `query-chunk` and `cas`. It may happen that $data$ is inserted at that bucket, hence the check at line 15. If not, the algorithm returns to line 8 to try the next bucket. If $p_{(i,j)}$ *is* occupied, `find-or-put` checks if $data \in p_{(i,j)}$ (line 17). In that case, `found` is returned, otherwise the next iteration is tried.

4   Experimental Evaluation

We implemented `find-or-put` in Berkeley UPC, version 2.20.2, and evaluated its performance by measuring the latency and throughput under various configurations. We compiled the implementation using the Berkeley UPC compiler and gcc version 4.8.2, with the options `upcc -network=mxm -O -opt`. All experiments have been performed on the DAS-5 cluster [8], using up to 48 nodes, each running CentOS 7.1.1503 with kernel version 3.10.0. Every machine has 16 CPU
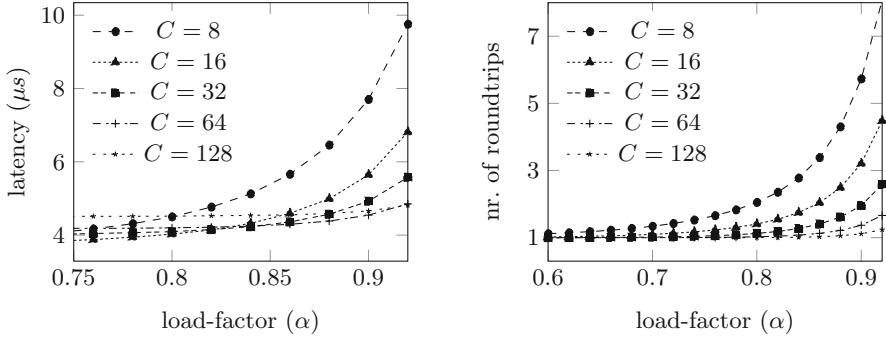
Fig. 2. The left plot shows the average latency of `find-or-put` (in microseconds) and the right plot shows the average number of roundtrips performed by `find-or-put` under different load-factors (α) and chunk sizes (C). Both plots show empirical data.

Table 1. The left table shows the latencies (in μs) and average number of roundtrips (Rt.) required by `find-or-put` to find a suitable bucket under various load-factors (α). The right table shows the total throughput ($\times 10^6$) of `find-or-put` under a *mixed* workload using various machines and processes per machine (Procs/M).

α	$C = 8$ Lat. Rt.	$C = 16$ Lat. Rt.	$C = 32$ Lat. Rt.	$C = 64$ Lat. Rt.	$C = 128$ Lat. Rt.	Procs /M	Machines 1	2	32	48
0.5	3.69 1.0	3.71 1.0	3.99 1.0	4.17 1.0	4.50 1.0	1	189.46	1.28	8.51	11.73
0.6	3.74 1.1	3.72 1.0	4.00 1.0	4.18 1.0	4.50 1.0	2	326.36	2.21	16.09	22.05
0.7	3.90 1.3	3.78 1.1	4.00 1.0	4.18 1.0	4.51 1.0	4	709.52	3.83	28.76	37.82
0.8	4.50 2.1	4.00 1.4	4.09 1.1	4.20 1.0	4.52 1.0	8	898.34	6.18	49.41	63.36
0.9	7.70 5.7	5.64 3.2	4.92 2.0	4.54 1.4	4.66 1.1	16	-	10.17	81.55	114.85

cores, 64 GB internal memory and is connected via a high-performance 48 Gb/s Mellanox Infiniband network. All experiments have been repeated at least three times and the average measurements have been taken into account.

4.1   Latency of `find-or-put`

We measured the latency of `find-or-put` using various chunk sizes while increasing the load-factor α. This is done by creating two processes on two different machines, thereby employing the Infiniband network. Both processes maintain a 1 GB portion of the hash table. The first process inserts a sequence of unique integers until α reaches 0.92, which appears to be our limit. The hash table started to return `full` when using 8-sized chunks and having $\alpha > 0.92$. The average latencies and the number of roundtrips have been measured at intervals of 0.02, with respect to α. These measurements are shown in Fig. 2 and Table 1.

The differences between latencies are very small for $\alpha \leq 0.5$, no matter the chunk size. For $\alpha = 0.5$, the average latency when using 64-sized chunks is 13 % higher compared to 8-sized chunks (shown in Table 1). However, the average
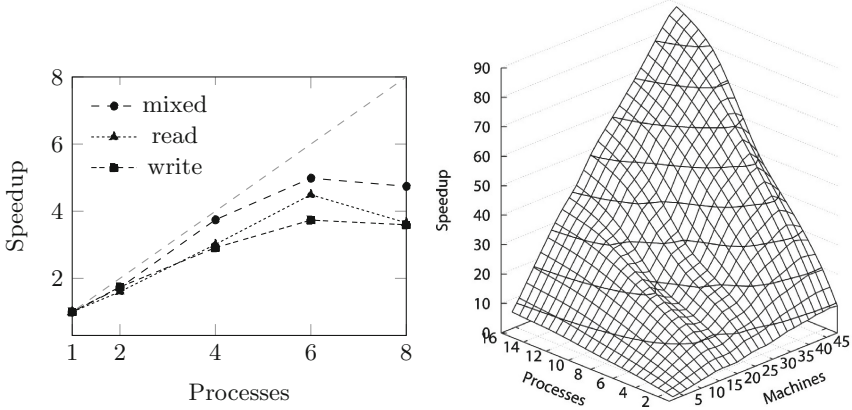
Fig. 3. Both plots show the average speedups with respect to the total throughput of `find-or-put`. The left plot shows the speedup with 1 machine, scaling from 1 to 16 processes. The right plot shows the speedup when scaling from 2 to 48 machines (i.e. using the Infiniband network) and scaling from 1 to 16 processes per machine.

latencies increase vastly for $C \geq 64$. For example, compared to 8-sized chunks, the latency is already 22 % higher for $C = 128$.

Moreover, the average latencies also increase vastly for $\alpha > 0.5$ when a low chunk size is used. By having small chunk sizes, more roundtrips are required by `find-or-put` to find the intended bucket, especially when $\alpha \geq 0.8$. By using a larger chunk size, higher load-factors are supported at the cost of slightly higher latencies. The average number of roundtrips directly influences the average latencies, which shows the importance of minimizing the number of roundtrips.

4.2   Throughput of `find-or-put`

The throughput of the hash table has been measured in terms of `find-or-put` operations per second (ops/sec). We scaled the number of machines from 1 to 48 and the number of processes per machine from 1 to 16. Each process owns a 1 GB portion of the hash table and inserts a total of 10^7 random integers. Three different workloads have been taken into account, namely:

- Mixed: 50 % finds and 50 % inserts
- Read-intensive: 80 % finds and 20 % inserts
- Write-intensive: 20 % finds and 80 % inserts

To get the proper find/insert ratio, each workload uses a different strategy to select the random integers. We used $C = 32$ and $M = 32$ in every configuration.

A subset of the measurements is shown in Table 1, and Fig. 3 shows speedups with respect to the *total* throughput, that is, the total sum of the throughputs obtained by all participating processes. In Fig. 3, the local speedups (left) are calculated relative to single-threaded runs. The remote speedups (right) are calculated relative to 2 machines, each having 1 process, thereby taking the Infiniband

network into account. Only throughputs under a *mixed* workload are presented, because the other workloads show very similar behaviour.

By comparing local throughput (i.e. using one machine) with remote throughput (i.e. using at least two machines), we observed a performance drop of several orders of magnitude. The local throughput reaches a peak of 8.98×10^8 ops/s. By using a mixed workload, the local throughput is up to 88 times higher than the peak-throughput obtained with two machines. A remote peak-throughput of 11.5×10^7 is reached, which is still 7.8 times lower than the local peak-throughput.

The local throughput reaches a speedup of 5x with 8 processes (see Fig. 3) under a mixed workload. We observed a vast decrease in local speedup when than 8 processes were used. However, when we use the Infiniband network, the performance remains stable, even when more than 8 processes per machine are used. The remote throughput reaches a speedup of 90x (with 48 machines, each having 16 processes) compared to 2 machines, each having 1 process. Compared to the single-threaded runs, a speedup of 0.61x is reached with 48 machines. This is expected; single-machine runs have better data-locality, as they do not use the network. Nonetheless, the entire memory of every participating machine can be fully utilized while maintaining good time efficiency.

4.3   Roundtrips Required by `find-or-put`

The average number of probes required by Pilaf during a key lookup in 3-way Cuckoo hashing with $\alpha = 0.75$ is 1.6 [10]. Nessie requires more roundtrips, since it uses 2-way Cuckoo hashing, which increases the chance on hash collisions compared to 3-way Cuckoo hashing. Our design requires only 1.04 probes on average for $C = 32$ and 1.006 probes for $C = 64$. Compared to Pilaf, this is an improvement of 53 % with 32-sized chunks.

Regarding the number of inserts, Pilaf is more efficient, as all inserts are handled by the server, at the cost of CPU efficiency. As part of the insertion procedure, a lookup must be performed to find an empty bucket. After that, the insert can be performed via `cas`, thereby requiring one extra roundtrip, in addition to the lookup operation. Therefore, our inserts are also more efficient than Nessie's inserts.

5   Conclusion

To build an efficient hash table for shared memory it is critical to minimize the number of roundtrips, because their waiting times contribute to higher latencies. The number of roundtrips is limited by the throughput of the RDMA devices. Lowering the number of roundtrips may directly increase the throughput.

Linear probing requires less roundtrips than Cuckoo hashing and Hopscotch hashing due to chunk retrievals, asynchronous queries, and the absence of relocations. Experimental evaluation shows that `find-or-put` can be performed in $4.5\,\mu$s on average with a load-factor of 0.9 for $C = 64$. This shows that the

hash table performance remains very high, even when the load-factor gets big. Furthermore, the entire memory of all participating machines can be used.

Table 1 shows that, in most cases, only one call to `query-chunk` would be enough for `find-or-put` to find a suitable bucket, especially for small values of α and large values of C. As future work, it would be interesting to dynamically determine the value of C to reduce the number of roundtrips. Moreover, we plan to use the hash table in a bigger framework for symbolic verification.

References

1. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Fourth Conference on Partitioned Global Address Space Programming Model. ACM (2010)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT press, Cambridge (2009)
3. Dragojevi, A., Narayanan, D., Hodson, O., Castro, M.: FaRM: Fast remote memory. In: 11th USENIX Conference on Networked Systems Design and Implementation, NSDI, vol. 14 (2014)
4. El-Ghazawi, T., Smith, L.: UPC: Unified Parallel C. In: ACM/IEEE Conference on Supercomputing. ACM (2006)
5. Farreras, M., Almasi, G., Cascaval, C., Cortes, T.: Scalable RDMA performance in PGAS languages. In: Parallel and Distributed Processing, pp. 1–12. IEEE (2009)
6. Herlihy, M.P., Shavit, N.N., Tzafrir, M.: Hopscotch hashing. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 350–364. Springer, Heidelberg (2008)
7. InfiniBand Trade Association: Accessed 9 May 2015. http://www.infinibandta.org
8. The Distributed ASCI Supercomputer 5 (2015). http://www.cs.vu.nl/das5
9. Kalia, A., Kaminsky, M., Andersen, D.G.: Using RDMA efficiently for key-value services. In: ACM Conference on SIGCOMM, pp. 295–306. ACM (2014)
10. Mitchell, C., Geng, Y., Li, J.: Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In: USENIX Annual Technical Conference, pp. 103–114 (2013)
11. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
12. Laarman, A., van de Pol, J., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Conference on Formal Methods in Computer-Aided Design, FMCAD, pp. 247–256 (2010)
13. Ross, K.A.: Efficient hash probes on modern processors. In: IEEE 23rd International Conference on Data Engineering, pp. 1297–1301. IEEE (2007)
14. Rumble, S.M., Ongaro, D., Stutsman, R., Rosenblum, M., Ousterhout, J.K.: Its time for low latency. In: HotOS (2011)
15. Szepesi, T., Wong, B., Cassell, B., Brecht, T.: Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In: First International Workshop on Rack-scale Computing (2014)
16. van Dijk, T., van de Pol, J.: Sylvan: Multi-core decision diagrams. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 677–691. Springer, Heidelberg (2015)

![Springer logo]