Efficient Usage of One-Sided RDMA for Linear Probing

Tinggang Wang\textsuperscript{*1}  Shuo Yang\textsuperscript{*1}  Hideaki Kimura\textsuperscript{2}  Garret Swart\textsuperscript{2}  Spyros Blanas\textsuperscript{1}

\textsuperscript{1} The Ohio State University  \textsuperscript{2} Oracle Corp.
\{wang.5264, yang.5229, blanas.2\}@osu.edu  \{hideaki.kimura, garret.swart\}@oracle.com

ABSTRACT

RDMA has been an important building block for many high-performance distributed key-value stores in recent prior work. To sustain millions of operations per second per node, many of these works use hashing schemes, such as cuckoo hashing, that guarantee that an existing key can be found in a small, constant number of RDMA operations. In this paper, we explore whether linear probing is a compelling design alternative to cuckoo-based hashing schemes. Instead of reading a fixed number of bytes per RDMA request, this paper introduces a mathematical model that picks the optimal read size by balancing the cost of performing an RDMA operation with the probability of encountering a probe sequence of a certain length. The model can further leverage optimization hints about the location of clusters of keys, which commonly occur in skewed key distributions. We extensively evaluate the performance of linear probing with a variable read size in a modern cluster to understand the trade-offs between cuckoo hashing and linear probing. We find that cuckoo hashing outperforms linear probing only in very highly loaded hash tables (load factors at 90% or higher) that would be prohibitively expensive to maintain in practice. Overall, linear probing with variable-sized reads using our model has up to 2.8\times higher throughput than cuckoo hashing, with throughput as high as 50M lookup operations per second per node.

1. INTRODUCTION

RDMA has been an important building block for many high-performance distributed key-value stores in recent prior work. Of particular interest are one-sided RDMA operations that allow reads and writes to remote memory and completely bypass the remote CPU. An RDMA READ operation specifies a contiguous memory region in the remote address space. The local NIC then transmits the request to the remote network adaptor (NIC), the remote NIC retrieves the requested memory region through a local DMA request and transmits the data back to the local NIC.

Cuckoo hashing has been widely adopted by RDMA-aware key-value stores \cite{3, 8, 20} due to the predictable and small number of RDMA operations to access a remote hash table. Prior work also proposed using one-sided RDMA verbs to implement linear probing by doing fixed-size lookups \cite{20}, where the client fetches a fixed number of hash slots per RDMA READ request and repeats the procedure until finding the key or the first empty slot. However, fetching a fixed number of slots per RDMA READ request may read too little or too much: in a lightly-loaded hash table, one should read very few slots to conserve network bandwidth; with high load factors, the average length of the probe sequence increases dramatically, hence one needs to read many more slots per RDMA READ request to reduce the number of round-trips in the network.

This paper introduces a mathematical model to optimize the read size when reading a remote hash table using linear probing with RDMA READ operations. The model takes into account the cost to complete one RDMA READ request for $R$ slots and the probability of completing the lookup after $k$ probes. We also propose to incorporate hints on the location of clusters of keys in the model, which is important when using datasets with a skewed key distribution.

We extensively evaluate the performance of linear probing with a variable read size in a modern cluster to understand the trade-offs between cuckoo hashing and linear probing. We find that cuckoo hashing outperforms linear probing only in very highly loaded hash tables (load factors at 90% or higher) that would be prohibitively expensive to maintain in practice. Outside of these extreme configurations, we find that both fixed-sized and variable-sized linear have better throughput than cuckoo hashing, and both complete a lookup at a latency that is about 60% of the latency of cuckoo hashing, even when cuckoo hashing probes all candidate locations in parallel. Fixed-sized linear probing has up to 1.6\times higher throughput than cuckoo hashing and variable-sized linear probing has up to 2.8\times higher throughput than cuckoo hashing. Variable-sized reads never perform worse than fixed-size reads, and are 1.7\times faster than fixed-sized reads in lightly loaded hash tables. In addition, our experiments show that storing records inline (aka. inside the hash table) should be preferred when the record size is up to 32 bytes. Records larger than 32 bytes should be stored out of band, for example in a heap, despite the need for additional RDMA requests to access the payload in this case. When using a skewed key distribution, using hints on the cluster locations returns matching records up to 1.7\times faster than when reading in a cluster-oblivious manner.

\* Contributed equally to this work.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and ADMS 2020.

11th International Workshop on Accelerating Analytics and Data Management Systems (ADMS’20), August 31, 2020, Tokyo, Japan.
To summarize, the core contributions of this paper are:

1. Introducing a mathematical model to balance the cost of initiating an RDMA READ request with the probability of completing linear probing in one round-trip.

2. Incorporating hints in the model on the locations of clusters of keys on the remote hash table, which commonly arises when some keys appear more frequently than others.

3. Performing an extensive experimental evaluation to understand the trade-offs between linear probing and cuckoo hashing for different load factors, inline or heap storage, as well as skewed key distributions.

2. BACKGROUND

2.1 Linear probing

Linear probing is an open addressing hashing scheme. Figure 1A shows an example of linear probing. When looking for key $c$, the probing algorithm starts from the slot determined by the hash function $h(c)$. In this example, the algorithm first checks slot $h(c) = 2$, and if it finds a matching key in this slot it returns it. If a matching key is not found, the linear probing algorithm reads the next slot until the matching key is found or an empty slot is reached. One disadvantage of linear probing is that the probing sequence can become very long when the hash table is highly loaded. Long probing sequences require checking more slots (or, in the worst case, the entire table) which takes more time to complete the probing.

When a client uses RDMA to access a remote hash table with linear probing, a trade-off arises when choosing how many slots each RDMA READ request will fetch. A client that fetches one slot per RDMA READ request, as when probing in local memory, will likely require multiple round trips to reach the key or an empty slot even when the table is moderately loaded. A client that retrieves multiple slots per RDMA READ request, on the other hand, is transferring unnecessary data and waiting longer for a larger RDMA READ to complete. A common solution in practice is to fix the size of each RDMA READ to a small multiple of the cache line size, such as 256 bytes, to balance the high cost of initiating an RDMA operation with minimal latency impact compared to shorter reads.

2.2 Cuckoo hashing

Cuckoo hashing uses a family of $k$ hash functions to find the location of a key, and guarantees that if the key exists it will be found in one of the $k$ slots, as determined by each hash function, regardless of how heavily loaded the hash table is. Figure 1B shows an example of a lookup operation in a 3-way cuckoo hash table. The lookup operation for key $c$ checks three slots $h_1(c) = 1$, $h_2(c) = 6$, and $h_3(c) = 4$. Prior work has also proposed using set associativity to further improve the space efficiency of cuckoo hash table [8]. With $m$-way set associativity, a cuckoo hash table stores $m$ records per slot. Each lookup operation to a hash slot checks all $m$ records in the slot for possible matches.

An one-sided RDMA-based implementation of a distributed hash table issues $k$ READ requests to access $k$ remote locations. There are two ways to issue RDMA READ requests for a $k$-way cuckoo hashing lookup: (1) sequentially, which inspects each candidate location in sequence and stops as soon as the key is found; and (2) in parallel, which issues $k$ RDMA READ requests concurrently. We refer to these variants as “Sequential Cuckoo” and “Parallel Cuckoo”, respectively, for the remainder of this paper. Parallel Cuckoo transmits $k$ RDMA READ requests in parallel, and hence hides the round-trip latency of multiple messages, whereas Sequential Cuckoo issues fewer RDMA READ requests if the key exists in the table.

3. VARIABLE READ SIZE MODEL

This section introduces a cost-based model that predicts the optimal read size. The model contains three parts. The first is modeling the cost of a single RDMA READ request that reads $R$ contiguous hash slots from a remote hash table. The second is a probability model that calculates the expected number of $R$-sized RDMA READ requests to reach the first empty slot given the expected load factor. Finally, we show how to further increase the prediction accuracy of the model if precise information about the occupancy of some slots in the hash table is known.

3.1 Predicting the read size

Assume that a RDMA READ request fetches $R$ contiguous hash slots at a time. Let $X(R)$ be a random variable that denotes the number of RDMA READ requests to reach the first empty slot, and let $T(R)$ be the cost of issuing a single RDMA READ request. The model predicts the read size $R^*$ that minimizes the total cost for the average probing sequence length, or

$$R^* = \arg \min_R \left( E[X(R)] \cdot T(R) \right)$$  \hspace{1cm} (1)$$

subject to the bandwidth constraint

$$R \cdot w \cdot \hat{\rho} \leq l$$  \hspace{1cm} (2)$$

where $w$ is the size of a slot (in bytes) in the hash table, $\hat{\rho}$ is the messaging rate, and $l$ is the bandwidth of the link.

The intuition is that it suffices to use a simple cost model based on the latency of a request, as long as the read size does not exceed the available network bandwidth. If the optimal read size would require more bandwidth the link can support, the model picks the largest size possible that will not saturate the link bandwidth to avoid queuing delays (which are not modeled).
the probability distribution function $P$ is that the messaging rate for R-sized messages as $\hat{\rho} = \rho_0 \cdot \frac{1}{h+w}$, where $\rho_0$ is the peak messaging rate for sending 0-byte payload messages [13], adjusted proportionally to the size of the request (in bytes) compared to a 0-byte payload message ($h$). The message size of a 0-byte RDMA READ request is $h = 30$ bytes based on the InfiniBand specification [11].

3.2 Modeling the cost of a single read $T(R)$

We use a latency model to capture the cost of issuing an RDMA READ request for $R$ contiguous slots. The cost of a single read is a linear function of the number of slots $R$ that are being retrieved:

$$T(R) = c + \alpha \cdot R \cdot w$$  (3)

Each RDMA READ request has a fixed cost $c$, which intuitively corresponds to the latency of initiating the request. The coefficient $\alpha$ is the signaling rate of the network in seconds per byte, which is reciprocal of the network speed. The variable $w$ corresponds to the size (in bytes) of a hash slot.

3.3 Computing the number of reads $X(R)$

Let $Prob_i(R)$ denote the probability of finding the first empty slot after probing $k$ slots, with $C_0 = 0$. Using exactly $i$ RDMA READ requests to find the first empty slot means the previous $(i-1)$ RDMA READ requests all failed to find an empty slot, and the first empty slot is found in the $i$-th RDMA READ request. Figure 2 visually shows how $Prob_i(R)$ can be computed. Performing $i$ RDMA READ requests for $R$ slots retrieves $i \cdot R$ slots in total. The probability of encountering the first empty slot after $i$ requests is $C_i \cdot R$, so $Prob_i(R)$ can be calculated as:

$$Prob_i(R) = C_i \cdot R - C_{(i-1)} \cdot R$$  (5)

We calculate the cumulative distribution function $C_k$ from the probability distribution function $P_k$, which is the probability that the first empty slot is exactly $k$ slots ahead from the location where probing started. Then $C_k$ is:

$$C_k = \sum_{i=0}^{k-1} P_i$$  (6)

For a hash table with $M$ slots of which $N$ are occupied, Knuth [15] calculates $P_k$ as follows:

$$P_k = M^{-N} \left( g(M, N, k) + g(M, N, k+1) + \cdots + g(M, N, N) \right)$$  (7)

where

$$g(M, N, k) = \binom{N}{k} f(k+1, k) f(M-k-1, N-k)$$

$$f(M, N) = \left( 1 - \frac{N}{M} \right)^M$$

From Eq. 7 it follows:

$$P_k - P_{k-1} = -M^{-N} g(M, N, k-1)$$  (8)

From Eq. 6 and Eq. 8 we have:

$$C_k = kP_0 - M^{-N} \sum_{i=0}^{k-1} (k-1-i) g(M, N, i)$$  (9)

where by definition $P_0 = 1 - \frac{N}{M}$.

Computing $P_k$ strictly following the mathematical definition requires multiplying very small numbers $(M^{-N})$ with very large numbers $(M^N)$, which poses problems with respect to the numerical stability of the result. Our implementation (which is described in detail in Section 3) computes the multiplications in Eq. 8 by transforming them to additions of logarithms with base $M$.

3.4 Accommodating cluster hints

The model proposed so far is oblivious to whether any particular slot is filled or not. However, if certain slots have been visited recently, such information may be known by a
client from prior lookups. Although one can use the cluster-oblivious model in these cases as well, an interesting question is whether the model can accommodate hints about the locations of possible clusters to further improve performance.

As shown in the example in Figure 4, a cluster is defined as a sequence of filled slots that starts at offset $h_i$, spans $\text{len}_i$ filled slots, and finishes at an empty slot at offset $h_i + \text{len}_i$. Let $h^*$ be the location of the slot where the probing will begin. Let the next cluster start at index $h_i \geq h^*$ and span $\text{len}_i$ slots. $C_k$ can be more accurately computed as follows:

$$C_k = \sum_{0}^{k-1} P_j \left( \frac{(0 < k \leq h_i - h*)}{C_{h_i - h^*}} \right) \left( \frac{(h_i - h^* < k \leq h_i - h^* + \text{len}_i)}{1} \right) \left( \frac{(h_i - h^* + \text{len}_i < k)}{0} \right)$$

As shown in Fig. 3, the intuition is that knowing about a cluster at $h_i$ that spans $\text{len}_i$ slots allows one to accurately compute that the probability $P_k$ of finding an empty slot in the $[h_i, h_i + \text{len}_i)$ range is 0, and that the slot $h_i + \text{len}_i$ is assumed to be empty and hence $C_k = 1$ at that point.

### 3.5 Model bootstrapping

The only parameters that cannot be set intuitively in our model are (1) the constant cost of one RDMA READ request $c$ and (2) the peak messaging rate $\rho_0$. These parameters need to be set through a simple benchmarking process. The constant cost $c$ of one RDMA READ request can be approximated as the latency to process one RDMA message with a 0-byte READ request, while the peak messaging rate $\rho_0$ can be approximated as the maximum achievable throughput to transmit 0-byte RDMA READ requests. One can use standard RDMA testing tools such as qperf [9] to measure these values at the time of deployment.

## 4. IMPLEMENTATION

This section describes the implementation of lookup operations in the hash table based on one-sided RDMA verbs. Sections 4.1 and 4.2 describe the overview of the design and the general procedure of processing lookup operations. Section 4.3 describes the linear probing procedure, and Section 4.4 describes how tracking clusters is implemented with linear probing. Finally, Section 4.5 describes the lookup procedures using cuckoo hashing.

### 4.1 Design overview

**Hash Table Design:** Our implementation supports storing key-value pairs either inside the hash table (inline storage) or in a separate heap (out-of-band storage). We use Knuth’s hash function to determine which slot a key is hashed to [10]. We assume that all keys are nonzero, and use zero to denote an empty slot. When using inline storage, each slot of the hash table stores the actual key-value pair. When storing records in the heap, each hash table slot stores one key signature and one offset. The offset of a filled slot can never be zero; an offset of zero implies that the slot is empty.

Figure 4 shows an example of how the key-value pair (“Oracle”, “Redwood City, CA”) is stored in the heap. The signature for key “Oracle” is 16 and the hash value of the key is 63. The heap offset is 28, which is the index of the actual key-value pair (“Oracle”, “Redwood City, CA”) in the heap.

We also implemented a 4-way set-associative table. When using inline storage, each slot of the hash table stores 4 records contiguously. When using a heap to store records, the 4 records are stored contiguously in the heap, as shown in Figure 5. Each hash table slot stores 4 signatures and one offset. In the example, offset 103 points to the address in the heap where 4 records are stored contiguously. Signature 85 corresponds to the key-value pair (“IRS”, “Austin, TX”) and signature 42 is for the key-value pair (“OSU”, “Columbus, OH”). The remaining two signatures are zero as the heap slots are empty.

### 4.2 General procedure of processing lookups

A lookup operation returns all matching key-value pairs, and transmits one or more RDMA READ requests. Algorithm 1 shows the general procedure of processing lookup operations, which is executed by multiple client threads independently. Each client thread maintains a queue called request_queue which buffers information about all outstanding RDMA READ requests. Each client processes multiple lookup operations simultaneously. First, the client schedules a lookup operation by pushing one or more RDMA READ requests into its request_queue. The client thread keeps posting RDMA READ requests in request_queue until it fills the send queue size of the connection (RDMA Queue Pair), and then waits until all posted requests are completed.
4.3 Linear probing with RDMA READ verbs

The linear probing algorithm varies in how it calculates the read size, or the number of slots fetched in one RDMA READ request. In fixed-size linear probing, the read size is fixed regardless of the load factor of the hash table. In variable-sized linear probing, the read size is determined from evaluating the model that was described in Section 4.3.

The fixed-sized and variable-sized linear probing share the same code path, since the read size can be regard as a parameter used when posting RDMA READ requests. The client schedules a lookup in linear probing by pushing a RDMA READ request to the request_queue to be posted in the next round. The procedures of scheduling new lookup operations and processing completed requests will be described in details in Section 4.3 and Section 4.5 for linear and cuckoo hashing, respectively.

4.4 Tracking clusters

Tracking clusters is exclusive for variable-sized linear probing lookups, and it shares a similar procedure compared with cluster oblivious linear probing lookups in Section 4.3. The difference is the client needs to record the information of m largest clusters and determines the read size for each lookup based on the cluster hints. The underlined parts in Algorithm 1 and Algorithm 2 shows how the client records and retrieves the information of clusters.

There can be multiple worker threads in a client probing the hash table. Each worker thread records the cluster information independently. To better use the idle CPU cycles when waiting for RDMA READ requests to be completed, we first buffer the discovered cluster information (line 9 and 10 in Algorithm 2), only update the data structure con-
taining the \( m \) largest clusters using the buffered cluster information after issuing all RDMA READ requests (line 15 in Algorithm 1).

The client determines the read size based on two cases: 1) If the hash value of the probing key \( h^* \) is located inside a cluster \([h_1, h_1 + \text{len}_i)\) (see Figure 5a), the client is certain about where the first empty slot is, so it issues one RDMA READ request to read till the empty slot; 2) If the hash value of the probing key \( h^* \) is not located inside any cluster \([h_1, h_1 + \text{len}_i)\), the client is not certain about where the first empty slot is, hence it uses the read size based on Section 3.4.

4.5 Cuckoo hashing with RDMA READ verbs

We implemented cuckoo hashing to compare the performance with that of linear probing. The client needs to issues \& RDMA READ requests to check the candidate locations determined by the \( k \) hash functions for probing a \( k \)-way cuckoo hash table. According to how the \( k \) RDMA READ requests are issued, there are two variants of cuckoo hashing lookups: 1) Sequential Cuckoo, in which a lookup issues the RDMA READ request checking the next candidate location only after the previous candidate location is checked; 2) Parallel Cuckoo, in which a lookup issues all \( k \) RDMA READ requests together checking all candidate locations when it is scheduled.

The client follows Algorithm 1 to schedule the lookups. To schedule a lookup for Sequential Cuckoo, the client pushes one RDMA REQUEST to \( \text{request\_queue} \) to read the first candidate location; for Parallel Cuckoo, the client pushes \( k \) RDMA REQUESTS to \( \text{request\_queue} \) to read all candidate locations. The checking order of the \( k \) candidate locations for Sequential Cuckoo is random. Algorithm 3 shows the procedures of the processing completed RDMA REQUESTS of cuckoo hashing for both inline storage (procedure \text{PROCESS\_INLINE}) and out-of-band storage (procedure \text{PROCESS\_OUT\_OF\_BAND}).

**Inline Storage:** As the key-value pairs are stored in the hash table, the client returns the matching key-value pair directly in line 4 of Algorithm 3.

**Out-of-Band storage:** The client needs to first issue an RDMA REQUEST to fetch the key signature and offset from the hash table, then issues an RDMA REQUEST to the heap to fetch the corresponding key-value pair if the key signature matches (see lines 13-15 of Algorithm 3).

**Set Associativity:** With inline storage, each RDMA REQUEST issued to the hash table fetches 4 slots. With out-of-band storage, each RDMA REQUEST issued to the hash table fetches one slot which stores 4 key signatures and 1 offset (see Figure 1), and each RDMA REQUEST issued to the heap always fetches a single key-value pair.

5. EVALUATION

This section compares linear probing to cuckoo hashing and evaluates the variable read size model to answer the following questions:

5.2 What are the latency percentiles to complete a lookup operation with cuckoo hashing and linear probing for different hash table load factors?

5.3 How sensitive is the model to misconfigured load factor?

5.4 How much faster can a client retrieve matching records by tracking clusters when the key frequency distribution is uniform?

5.1 Experimental setup

**Platform:** All experiments were conducted in a shared, 56-node cluster where every machine is equipped with 2 Intel Xeon E5-2680 v4 CPUs and one Mellanox ConnectX-5 EDR InfiniBand adaptor card. Every machine runs CentOS Linux release 7.4.1708 (Core). All experiments used two machines: one server and one client.

**Workload:** We generated hash tables with \( 120 \times 2^{20} \) records and load factors that range from 0.25 to 0.95. Hence, the total table size varies based on the load factor, ranging from about \( 126 \times 2^{20} \) slots for the 0.95 load factor to
480 × 2^{20} slots for 0.25 load factor. The cuckoo hash table is configured as a 3-way cuckoo hash table, which has been shown to balance memory efficiency and performance in prior work [20]. The record size ranges from 8 bytes to 128 bytes: the key size is fixed to 4 bytes, and the value size varies from 4 bytes to 124 bytes depending on the experiment. For each load factor, we pre-generated 10 different hash tables storing random records.

**Operation:** In every experiment, the server randomly picked one of the 10 pre-generated hash tables for the tested load factor. The server then allocated and registered memory for RDMA accesses, and loaded the table into RDMA-accessible memory. The server then waited passively until all client operations are finished to close all connections, release the memory and shut down. The client spawned one worker thread per CPU core for a total of 28 threads. Each worker thread initialized one queue pair, connected with the server and started probing. We used `pthread_barrier` to make sure all threads start and stop probing at the same time. Each worker thread probed and returned all matching records for a randomly chosen key. All worker threads probed records for at least 100 seconds in each experiment. All experiments were repeated at least 5 times. We report averages from all runs, and also plot 95% confidence intervals when the variability is significant.

In variable-sized linear probing, the number of bytes that will be retrieved in each RDMA READ request is determined from the analytical model (see Section 3.5), and changes based on the load factor. Fixed-size linear probing retrieves the same number of bytes regardless of the hash table load factor. Conventional wisdom suggests retrieving a few cache lines per RDMA READ request, as retrieving less data will not improve the latency of the RDMA operation. We thus configured the read size of the fixed-size linear probing algorithm to be 256 bytes in our experiments.

**Network configuration:** Both the client and the server have one queue pair per CPU core, or 28 queue pairs in total. When initializing a queue pair, the maximum number of outstanding RDMA REQUESTS on this connection can be configured by changing the `max_dest_rd_atomic` and `max_rd_atomic` parameters. Prior work uses different settings: DrTM [4] and online tutorials [2] configure 1 outstanding RDMA REQUEST per queue pair, while FaSST configures 16 outstanding requests [14]. We observed that the lookup throughput increased by as much 4× after changing the maximum outstanding RDMA REQUESTS from 1 to 16 (where 16 is the maximum number our hardware can support). We thus set the number of outstanding RDMA REQUESTS to 16 to utilize the maximum parallelism between requests.

Another important configuration parameter is the size of the send queue in every queue pair. We measured the throughput of lookup operations, the send queue size was set to the maximum value supported by our hardware (8192 requests) to simulate the highest possible level of offered load. When we measured the latency of lookup operations, the send queue size was set to the minimum number of RDMA REQUESTS that are needed per lookup (that is, 1 for Linear probing and Sequential Cuckoo, and 3 for Parallel Cuckoo) to simulate an idle system.

**Model parameters:** The parameters for Eq. 2 and Eq. 3 are set as follows in our experiments: The link bandwidth $l$ is 100Gbps, which is the InfiniBand EDR bandwidth, and hence the coefficient $\alpha$ (signalling rate) is 0.08 ns/byte. When using inline storage, the slot size $w$ equals the record size (8, 32 or 128 bytes, respectively). When storing records in the heap, each hash table slot in linear probing has one 1-byte key signature and 4-byte offset, so $w = 5$ bytes. Following the model bootstrapping procedure described in Section 5.3 the peak messaging rate $p_0$ was 87.17 million RDMA REQUESTS per second and the constant term $c$ (fixed cost of initiating a new RDMA request) was 1,290 ns. We built a simple benchmarking program for this purpose that uses the same network configuration as described earlier in this section.

### 5.2 Different cuckoo hashing variants

As presented in Section 4, probing in cuckoo hashing can be done sequentially (Sequential Cuckoo) or in parallel (Parallel Cuckoo). In addition, the associativity of the table can be configured: one can allow multiple records in one slot or can allow only one record per hash slot. This experiment compares the lookup throughput of cuckoo hashing using different design choices with 8-byte records.

Figure 6 plots the throughput of four different load factors. For both 4-way associative and non-associative hash tables, the throughput difference between Parallel Cuckoo and Sequential Cuckoo is not statistically significant at the 5% level across all load factors. The throughput difference between a non-associative and a 4-way set-associative cuckoo hash table is statistically significant, but on average the throughput of 4-way set-associative cuckoo hash tables is only 3% lower. However, associativity is very important for resolving collisions when creating a cuckoo hash table at high load factors: note that Figure 6 does not show results from non-associative hash tables at 95% load, because pre-generating the tables was taking impractically long (weeks) to complete due to the very long displacement sequences for every insertion. (This finding is corroborated in prior work that points out that insertion is prohibitively expensive in non-associative hash tables with load factors greater than 50% [8].) The remainder of this evaluation uses 4-way set-associative hash tables for all experiments with cuckoo hashing.

![Figure 6](image-url)
Figure 7: Lookup throughput of parallel cuckoo hashing with 4-way set associativity, fixed-sized linear probing, and variable-sized linear probing (with and without the bandwidth constraint of Eq. 2).

5.3 Linear probing vs. cuckoo hashing

This experiment compares the lookup throughput in linear probing and cuckoo hashing to understand trade-offs between probing in different hash tables and how throughput changes at different load factors. We used 8-byte records in this experiment. Figure 7 plots the lookup throughput of linear probing and cuckoo hashing at different load factors. For cuckoo hashing, we only show the Parallel Cuckoo variant (labeled "Cuckoo") as the throughput difference of different variants is not statistically significant (see Section 5.2). For variable linear probing, the dotted line labeled "Var. Linear (uncons.)" shows the throughput if the model were not bound by the bandwidth constraint (Eq. 2). Figure 7 shows that the throughput of variable-sized linear probing was up to 2.8× higher than cuckoo hashing and up to 1.7× higher than fixed-size linear probing, except for very highly loaded hash tables. Cuckoo hashing only outperformed linear probing at very high utilization (90% and above), but heavily loaded tables would be unrealistic configuration choices in practice due to the prohibitively large number of record displacements on every update to the hash table.

To provide additional insights into the throughput of the algorithms, we show the number of RDMA requests per lookup and the number of slots retrieved per RDMA READ for different linear probing variants in Table 1. In addition, we show the effective bandwidth utilization (that is, the bandwidth used to transmit the message payload only) for each algorithm in Figure 8. Cuckoo hashing was always limited by the messaging rate and not the bandwidth because it transmitted 3 RDMA requests per lookup regardless of the load factor. In lightly loaded hash tables, the weakness of fixed-size linear probing was reading too much data: fixed-size linear probing used almost 8 GB/sec of bandwidth (see Figure 8) and almost never issued more than one RDMA request at 0.25 and 0.5 load factors. The efficiency of fixed-size linear probing quickly deteriorated at high load: when the hash table was 95% full, 48% of the probes required 4 or more RDMA requests, and 13% of the probes required 16 or more RDMA requests.

Variable-sized linear probing increases the read size as the load factor increases to accommodate longer expected probe sequences, as shown in Table 1. Changing the read size according to the analytical model without any bandwidth constraint ("Var. Linear (uncons.)") resulted in 89% or more of all lookups needing only 1 RDMA READ request across all load factors. However, as shown in Table 1, the read size rapidly increases from 5 slots to hundreds of slots at load factors of 80% and higher. This makes the unconstrained variable linear probing algorithm limited by the network bandwidth (see Figure 7), which is 12.5 GB/sec for InfiniBand EDR; throughput degrades by up to 2× over fixed-size linear probing (see Figure 7). By imposing the bandwidth constraint, variable-sized linear probing retains its good performance at low load factors and limits the read size to not saturate the network at high load.

5.4 Latency of different probing strategies

We measured the latency to complete a lookup operation with different probing strategies. In this experiment the record size is 8 bytes and each thread worker only issues one lookup operation at one time. Figure 7 shows the latency for cuckoo hashing and linear probing when the load factor ranges from 25% to 95%.

As expected, the latency of the cuckoo hashing variants was not sensitive to the load factor, as cuckoo hashing performed the same number of RDMA requests regardless of the load factor. Comparing the two cuckoo hashing variants, checking the 3 possible locations sequentially (Sequen-
Figure 9: Latency for different load factors for linear probing and cuckoo hashing. The colored bar shows the median latency, the solid vertical line shows the interquartile range (25th-75th percentile), and the error bars show the 10-th and 90-th percentile of latency.

Figure 10: The measured throughput and the predicted cost for different read sizes for 50% and 85% load factors.

pared to picking other read sizes: for both 50% and 85% load factors, the observed throughput was at the peak of the curve in Figure 10a. The difference in the throughput between the chosen read size and read size 7 for load factor 50%, as well as the chosen read size and read sizes 18 and 32 for load factor 85% is not statistically significant at the 0.05 level.

5.6 Model sensitivity to the load factor

One question is how sensitive is the reported throughput of variable-sized linear probing to different load factors, as the actual load of a hash table may be different than assumed or sensed remotely through a sampling process. Of particular interest is lightly-loaded hash tables where the configured read size can vary by as much as 4×, as shown in Table 1 under column group “Read size (slots)”. (A welcome side-effect of the bandwidth constraint is that it equalizes the read size configuration at high load factors.)

Table 2 shows the throughput of variable-sized linear probing if the load factor assumed by the model is different from the actual load factor. The row names are the actual load factors and the column names are the load factor assumed by the model; the cell reports the lookup throughput under the condition determined by the row and column. The diagonal cells show the lookup throughput when the actual load factor is the same as the load factor assumed by the model, and correspond to the “Variable Linear” line in Figure 9. The lookup throughput will drop by at most 20% (when assuming load factor of 65% but encountering 50%) if the model mistakenly picks the neighboring load factor among the three load factors in the table. We conclude that in

### Table 1: Lookup throughput (millions/second)

<table>
<thead>
<tr>
<th>Assumed load</th>
<th>Actual load 0.25</th>
<th>Actual load 0.50</th>
<th>Actual load 0.65</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>50.8</td>
<td>43.0</td>
<td>34.5</td>
</tr>
<tr>
<td>0.50</td>
<td>42.3</td>
<td>34.2</td>
<td>35.1</td>
</tr>
<tr>
<td>0.65</td>
<td>31.0</td>
<td>39.2</td>
<td>35.2</td>
</tr>
</tbody>
</table>
lightly-loaded hash tables variable-sized linear probing has better throughput than fixed-size linear probing (cf. “Fixed Linear” in Figure 7) even when the load factor is significantly misconfigured.

5.7 Out-of-band storage

Storing records outside the hash table improves the locality of the index structure and becomes appealing as record sizes increase. To understand when one should use out-of-band storage, we ran an experiment comparing the lookup throughput of probing records of different sizes in the hash table (inline storage) and in a separate heap (out of band storage). This experiment uses a load factor of 65% and varies the record size from 8 bytes to 128 bytes.

Figure 11 shows the lookup throughput for cuckoo hashing (parallel variant), fixed-sized linear probing and variable-sized linear probing for both inline and out of band storage. Our first observation is that at this load factor the best-performing algorithm is variable-sized linear probing for all record sizes and both inline and out of band storage. (This generalizes the result first seen in Figure 7). The throughput when storing records inline is very sensitive to the record size: the throughput of the best-performing algorithm, Variable Linear, drops from 36 M lookups per second to 8 M lookups per second when the record size increases from 8 to 128 bytes. The throughput is less sensitive to changes in the record size for out-of-band storage: the best-performing algorithm, Variable Linear, only drops from 23 M lookups per second to 19 M lookups per second when the record increases from 8 to 128 bytes.

Linear probing is more sensitive to changes in the record size than cuckoo hashing. This sensitivity is due to the different number of slots the linear probing algorithms will retrieve at different record sizes. As shown in Table 3, for 8-byte records, all algorithms were reading 2-6 slots per RDMA READ. When the data is stored out of band, the slot in the hash table only takes 5 bytes (4-byte offset and 1-byte key signature, as explained in Section 4.4), and hence one RDMA request will retrieve 30-50 slots. As shown in Table 3 this allows the linear probing algorithms to almost always complete in a single lookup to the hash table with out-of-band storage.

The trade-off for out-of-band storage is making RDMA READ requests to the hash table more efficient (smaller hash slots) in exchange for additional RDMA READ requests to the heap (to retrieve the payload). Take the variable-sized linear probing as an example. When record size is 128 bytes, probing records in out-of-band storage means retrieving 623 fewer bytes per RDMA READ request to the hash table compared to the case of inline storage. Although one needs about 1.04 additional RDMA READ requests to the heap (see Table 3) compared with Inline storage, the overall lookup throughput of probing records stored in the heap is 2.4× higher than when storing them inline. Overall, the results show that out of band storage will result in higher throughput when the record size is 32 bytes or more.

5.8 Cluster tracking

This experiment investigates the effectiveness of tracking clusters for variable-sized linear probing when the keys in the hash table have a skewed frequency distribution. In this experiment, 8-byte records are stored in a 65% loaded hash table, and the 4-byte keys follow the Zipf distribution. The read size is 23 slots for cluster-oblivious reads, as computed by the model (Eq. 8 and Eq. 9). When tracking clusters, each thread maintains the 1000 largest clusters it has encountered so far. The experiment starts with an empty cluster tracking set and the implementation tracks the 1000 largest clusters at runtime. (See Section 4.4 for details.)

Figure 12 shows the rate of reading matching records with and without cluster tracking for different levels of skew. (This experiment reports the rate of reading matching records instead of the lookup throughput because the duration of a single lookup varies with a skewed key distribution based on the number of matching keys a lookup finds.) Tracking clusters improves the rate of finding matching records up to 1.7× compared with cluster-oblivious reading when the keys

<table>
<thead>
<tr>
<th>Record size (bytes)</th>
<th>Read size (slots)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>Any</td>
</tr>
</tbody>
</table>

Table 3: Number of slots retrieved per RDMA READ request for different hashing methods for inline and out-of-band storage. Load factor is 65%.
Figure 12: Rate of finding matching records for variable-sized linear with the cluster-oblivious model and when tracking clusters at 65% load factor and varying skew.

Figure 13: The read size chosen by the cluster-tracking model at different read offsets from the start of a cluster. The cluster in this example starts at $x = 0$ and spans 10 slots until $x = 9$.

6. RELATED WORK

Many researchers have studied cuckoo hashing. Alcantara et al. [1] studied using cuckoo hashing on GPUs. Li et al. [16] proposed the idea of Multi-copy Cuckoo to address the issue of collision resolution for cuckoo hashing at high load factors. CUCKOO SWITCH [24] is a software-based Ethernet switch adopting cuckoo hashing. Silt [17] and MemC3 [8] adopted cuckoo hashing for their systems.

Focusing on distributed hash tables, Pilaf also proposes a basic design that uses linear probing, which keeps fetching one slot per RDMA READ request from the hash table and using another RDMA READ request fetching the key-value pair in the corresponding offset if the slot is not empty. This procedure is repeated until the requested key is found. FaRM [7] also uses a heap to store key-value pairs separately.

Nessie [3] adopts a symmetric system architecture in which each node contains a cuckoo hash table and a heap storing actual key-value pairs. A hash table entry can point to a key-value pair stored in a remote heap. Nessie uses one-sided RDMA READ request to fetch the data entries and index entries if stored remotely. In Nessie, a lookup contains a forward pass to search for data entry and a reverse pass to verify consistency due to the potential concurrent insert and delete operations.

MICA [15] uses a chaining hash table to locate key-value pairs in a log structure. In lossy mode, the oldest entry in the hash table will be evicted when inserting a key to a full bucket. MICA adopts exclusive access in which each CPU core can only access its assigned partition of data. A client sends an RPC-based request via Intel’s Data Plane Development Kit (DPDK) the server, and the request will be re-direct to the CPU core that can access to the corresponding partition.

The key-value stores of HERD [12] and FaSST [14] are based on MICA, and both use RDMA-based RPCs for requests and data exchange between nodes. HERD’s RPCs combine both one-sided and two-sided RDMA verbs: a client issues requests to the server by writing to server’s buffer using one-sided RDMA WRITES, and the server sends back results via two-sided RDMA SEND. FaSST uses two-sided RDMA SEND/RECV requests to exchange packets containing the requests and the data, and applies optimizations such as co-routines and doorbell batching.

Hopscotch hashing [18] is a hashing scheme that also requires a small and constant number of probes to find a key. FaRM [7] adopts a variant of hopscotch hashing by combining chaining and associativity due to the trade-off between space efficiency and the size and number of one-sided
RDMA READ requests used for lookups. For a lookup, it first performs a one-sided RDMA READ request to fetch two buckets containing $H$ slots in total, where $H$ is the size of the neighborhood of hopscotch hashing that guarantees to contain the lookup key. If the lookup fails, it continues searching along the linked chain of overflow blocks. Deb Nath et al. showed hopscotch hashing has better performance than cuckoo hashing for write-intensive workload over phase change memory (PCM).

RDMA is not only used for key-value stores but also distributed Online Transactional Processing (OLTP) systems. DrTM [4] combines HTM and RDMA for distributed transactions. DrTM+H [24] proposes that a hybrid of one-sided and two-sided RDMA verbs is better for distributed OLTP systems. NAM-DB [24] proposes another design for distributed transaction processing that leverages RDMA.

The design underpinnings of our system follow best practices in prior work. Ruhela et al. [21] emphasized the overlap of computation and communication is critical for good performance, and in our work we updated the data structure maintaining the cluster set only when waiting for the RDMA READ requests to be completed. Other works have also shown how to better use RDMA [13, 19], as well as use RDMA for other data structures, such as trees [20]. In our design, the lookups being processed in a batch do not necessarily belong in the same processing stage: lookups in the same batch are not synchronized, they may not be issuing the same number of RDMA READ requests, and some lookups may target the heap while others target the hash table. This idea was extended from the asynchronous progress design of DART [23].

7. CONCLUSION AND FUTURE WORK

This paper studies using one-sided RDMA verbs to to probe remote hash tables, and finds that linear probing with fixed-size reads either reads too little and hence needs many round-trips at high load factors, or reads too much and wastes network bandwidth at low load factors. The paper investigates variable-sized linear probing that relies on an analytical model to pick the read size. Using this model, linear probing with variable-sized reads achieves up to 1.7x higher throughput and never performs worse than fixed-size reading; variable-sized linear probing also outperforms cuckoo hashing by as much as 2x for load factors less than 90%. The results also show lookups for small records (less than 32 bytes) have better performance when data is stored inline (inside the hash table), while records larger than 32 bytes are better suited for storing out of band in a heap. Finally, we also show how to augment the variable-sized linear model with cluster hints from prior probes, which works especially well for skewed key distributions as it improves the key matching rate by up to 1.7x.

The results point to a number of promising avenues for future work. First, the comparison between inline and out of band storage suggests that robust lookup performance for workloads with variable-length records can only be achieved through a hybrid layout, where the insertion algorithm automatically decides whether to place a record inline or out of band. Second, the results show that cuckoo hashing transmits multiple small messages and uses bandwidth very judiciously. This is both a blessing and a curse: cuckoo hashing will perform well in bandwidth-constrained settings, but cannot fully utilize the network bandwidth with small messages. In contrast, linear probing requires substantial bandwidth as the load factor approaches or exceeds 80%. A promising direction for future work is to use two-sided send/receive communication to conserve network bandwidth with linear probing in highly-loaded hash tables. The relative performance of two-sided linear probing over one-sided cuckoo hashing at high load factors is an open question. Finally, a limitation of the InfiniBand verbs interface is the requirement to access contiguous remote memory in each request, which means that independent remote reads require multiple RDMA requests. Other network interfaces, such as Cray DMAPP [5], can access non-contiguous remote locations in a single request through a gather network primitive. There is unexplored potential to substantially improve the performance of independent reads, both within and across lookups, with a gather primitive in future work.

8. ACKNOWLEDGEMENTS

This material is based upon work that was partially supported and funded by Oracle America, Inc. This work was supported in part by National Science Foundation grant SHF-1816577.

9. REFERENCES