Balancing CPU and Network in the Cell Distributed B-Tree Store

Christopher Mitchell Kate Montgomery Lamont Nelson Siddhartha Sen* Jinyang Li

New York University **Microsoft Research*

{cmitchell, kem493, lamont.nelson, jinyang}@cs.nyu.edu, sidsen@microsoft.com

Abstract

In traditional client-server designs, all requests are processed at the server storing the state, thereby maintaining strict locality between computation and state. The adoption of RDMA (Remote Direct Memory Access) makes it practical to relax locality by letting clients fetch server state and process requests themselves. Such client-side processing improves performance when the server CPU, instead of the network, is the bottleneck. We observe that combining server-side and client-side processing allows systems to balance and adapt to the available CPU and network resources with minimal configuration, and can free resources for other CPU-intensive work.

We present Cell, a distributed B-tree store that combines client-side and server-side processing. Cell distributes a global B-tree of "fat" (64MB) nodes across machines for server-side searches. Within each fat node, Cell organizes keys as a local B-tree of RDMA-friendly small nodes for client-side searches. Cell clients dynamically select whether to use client-side or server-side processing in response to available resources and the current workload. Our evaluation on a large RDMA-capable cluster show that Cell scales well and that its dynamic selector effectively responds to resource availability and workload properties.

1 Introduction

In the traditional client-server design, the server does the vast majority of the computation, storing state and processing operations by performing computation over the state. Clients simply send RPC requests and receive replies from the server. Maintaining strict locality of computation and state is crucial for performance when the cost of communication is high.

Commodity clusters have recently started to embrace ultra-low-latency networks with Remote Direct Memory Access (RDMA) support [3, 1, 7], just as in-memory storage has become practical and performant [34, 37, 31, 45]. RDMA is supported over InfiniBand or over Ethernet via RoCE [46, 49], each of which offers high throughput (20-100 Gbps) and low round-trip latency (a few microseconds). With RDMA, a machine can directly read or write parts of a peer's memory without involving the remote machine's CPU or the local kernel; traditional message passing can also still be used.

As a result, RDMA has drastically lowered the cost of communication, thereby permitting an alternative system design that relaxes locality between computation and state. Clients can process requests by fetching server state using RDMA and performing computation on the state themselves [7, 32]. Client-side processing consumes similar total CPU resources to server-side processing, except with the CPU load shifted to the clients. However, this flexibility comes at a cost: fetching server state consumes extra network resources, which may become a bottleneck. For datacenters with capable networks, the bottleneck network resource is each server's NIC(s).

Several existing in-memory distributed storage systems utilize RDMA-capable networks; all are unsorted key-value stores that exclusively use client-side or server-side processing. For example, Pilaf [32] and FaRM [7] rely on client-side processing for all read operations. HERD [22] uses only server-side processing. None of these solutions is satisfactory: we recognize that practical in-memory storage systems exist on a continuum between being CPU-bound and network-bound that also shifts as workloads change. Therefore, we explore a hybrid approach that augments server-side processing with client-side operations whenever bypassing the server CPU leads to better performance.

Modern bare-metal servers are usually equipped with as many CPU cores as necessary to saturate the server's NIC [2]. Nevertheless, there are several common scenarios where servers' CPUs can become bottlenecked, causing client-side processing to be more desirable. First, when deploying a distributed storage system in the cloud, the virtual machines' CPUs must be explicitly rented. As a result, one usually reserves a few CPU cores that are sufficient for the average load in order to save costs. Doing so leaves servers overloaded during load spikes[1]. Second, a networked system may be deployed in a shared cluster where the same physical machines running the servers also run other CPU-intensive jobs, including storage-related application logic, to maximize cluster utilization. In this case, one also does not want to assign

[1]One can dynamically add more server instances on-the-fly in case of server overload. However, this would be too slow to handle load spikes of a few seconds.

more CPU cores than necessary for the servers to handle the average load, thereby also resulting in server CPU overload during load spikes.

In this paper, we investigate how to balance CPU and network by building a distributed, in-memory B-tree store, called Cell. We choose a B-tree as a case study system because of its challenges and importance: as a sorted data structure, B-trees serve as the storage backend for distributed databases. A distributed B-tree spreads its nodes across many servers, and supports get, put, delete, and range operations. The key component underlying all operations is search, i.e. traversing the B-tree: Cell combines client-side and server-side processing for B-tree searches. Cell builds a hierarchical B-tree: Cell distributes a B-tree of "fat" nodes (*meganodes*), each containing a local B-tree of small nodes, across server machines. This hierarchical design enables simultaneous server-side search through meganodes and efficient client-side search through small nodes. Our architecture permits reliable lock-free search with caching even in the face of *concurrent* writes and structural changes to the tree. Our Cell prototype also provides distributed transactions on top of the B-tree using traditional techniques; we omit a discussion of this feature due to scope.

In order to arrive at the best balance between using client-side and server-side processing, Cell needs to dynamically adjust its decision. Ideally, when server CPUs are not saturated, Cell clients should choose server-side searches. When a server becomes overloaded, some but not all clients should switch to performing client-side searches. The goal is to maximize the overall search throughput. We model system performance using basic queuing theory; each client independently estimates the "bottleneck queuing delays" corresponding to both search types, then selects between server-side and client-side search accordingly. This dynamic selection strategy achieves the best overall throughput across a spectrum of different ratios of available server CPU and network resources.

We have implemented a prototype of Cell running on top of RDMA-capable InfiniBand. Experiments on the PRObE Nome cluster show that Cell scales well across machines. With 16 server machines each consuming 2 CPU cores, Cell achieves 5.31 million ops/sec combining both server-side and client-side searches, 65% faster than server-side search alone while still leaving the remaining 6 cores per machine for CPU-intensive application logic. More importantly, Cell balances servers' CPU and network resources, and is able to do so in different environments. Cell clients make good dynamic decisions on when to use client-side searches, consistently matching or exceeding the best manually-tuned fixed percentages of client-side and server-side searches or either of the search types alone. The system responds quickly (in $< 1s$) and correctly to maintain low operation latency in the face of load spikes and tree structure modifications.

We present Cell's design in Section 2 and describe implementation-specific details in Section 3. We thoroughly evaluate Cell's performance and design choices in Section 4. We explore related systems in Section 5.

2 Cell Design

Cell provides a sorted in-memory key-value store in the form of a distributed B-tree. In this section, we give an overview of Cell (2.1) and then discuss the main components of our design: our B-tree structure (2.2) and our hybrid search technique (2.3, 2.5).

2.1 Overview

In designing the Cell distributed B-tree store, we make two high-level design decisions: (1) support *both* client-side and server-side operations; (2) restrict client-side processing to read-only operations, including B-tree searches and key-value fetches. (1) is made possible by a hierarchical B-tree of B-trees. (2) is logical because practical workloads are search- and read-heavy [34], and client-side writes would involve much more complexity. As Section 4.5 shows, we can reap the benefits of client-side processing across a variety of workloads with different fractions of read vs. write operations. Cell's basic design faces two novel challenges: how to ensure the correctness of RDMA searches during concurrent server-side modification, and when should clients prefer client-side over the default server-side processing? Although a large body of existing work explores concurrent B-trees, these works assume that servers' CPUs still have full control over access to the servers' data. With RDMA, the storage system must provide its own techniques to synchronize reads and writes, or at least ensure that reads are performed over consistent data.

Cell organizes data in a hierarchical B-tree of B-trees to ensure that both server-side and RDMA-based searches are efficient. At the cluster level, Cell builds a B-tree out of fat nodes called *meganodes*, containing tens or hundreds of megabytes of structural metadata. Meganodes are spread across the memory of the servers in the cluster. Within each meganode, Cell builds a local B-tree consisting of small nodes (e.g. 1KB). The local B-tree allows a server to search efficiently for a key within a meganode, while simultaneously allowing remote clients to search within a meganode using a small number of efficient RDMAs. Other systems like BigTable [5] only support server-side search, so they can use different searchable local data structures such as skiplists that would require many more roundtrips to access remotely.

We adopt the common practice of storing the data of a B-tree at its leaf level. Thus, the leaf meganodes of the tree store local pointers to data while the internal megan-

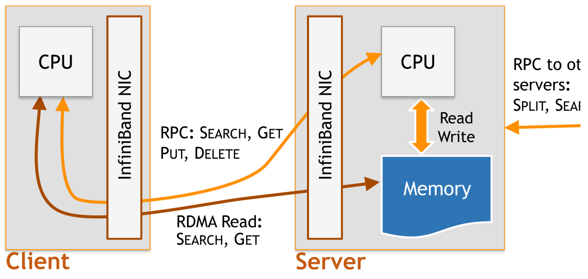

Figure 1: Cell's architecture and interactions (Section 2).

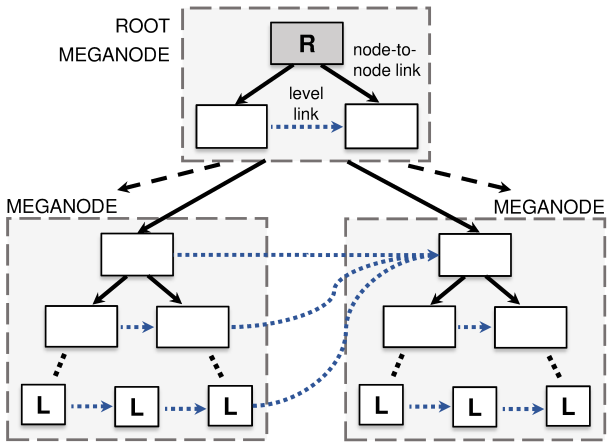

Figure 2: The structure of Cell's data store, a B-link tree of B-link trees. Each individual meganode contains a complete level-linked tree; the meganodes are also level-linked. The root meganode contains the root node (R). The leaves (L) of the bottom meganodes point to key-value data stored in the local extents region of that machine.

odes store remote pointers to other meganodes on remc machines. All pointers consist of a region ID and offs Each server's in-memory state includes multiple lar contiguous memory areas (e.g. 1GB) containing node regions that hold meganodes or extents regions holding key-value data. All clients and servers maintain a cache of the global region table that maps region IDs to the IP addresses of the responsible machines. Servers also actively exchange region information with each other asynchronously. We assume that server membership is maintained reliably using a service like Zookeeper [18].

As shown in Figure 1, clients communicate with servers to perform B-tree operations including search (contains), get (read), put (insert/update), and delete. Of these operations, search and get may be performed by clients via RDMA. Servers also communicate with each other to grow and maintain the distributed B-tree. The coordination between servers adds a level of complexity not present in prior RDMA-optimized systems like FaRM [7] or Pilaf [32]. However, we minimize this complexity by carefully designing our B-tree, discussed next.

2.2 Server-Side B-tree Operations

Cell uses a type of external B-tree called a *B-link tree* [25]. We use the same structure at both the meganode scope and within each meganode, as illustrated in Figure 2. B-link trees offer much higher concurrency than standard B-trees due to two structural differences: each level of the tree is connected by right-link pointers, and each node stores a *max key* which serves as an upper bound on the keys in its subtree. (We also store a min key to cope with concurrent RDMA reads; see 2.3.) Sagiv [38] refined the work of Lehman and Yao [25] into an algorithm that performs searches lock-free, insertions by locking at most one node at a time, and deletions by locking only one node. The lack of simultaneous locking makes the algorithm well-suited to distributed and concurrent settings [19, 30].

We follow Sagiv's algorithm when operating within a meganode. A search for a key follows child pointers and, if necessary, right-link pointers until the correct leaf node is reached. Range queries are implemented by following right links at the leaf level. Insertions and deletions begin with a search to find the correct leaf node. If an insertion causes a leaf node L to split (because it is full), we lock L long enough to create a new node L' containing roughly half of L's contents, and set L's right-link pointer to L'. The right link ensures that concurrent searches can reach L' (guided by the max key of L) even if it has no parent yet. The split key is then inserted into the parent as a separate, decoupled operation that may propagate further up the tree. Deletions simply remove the key from its leaf node under a lock. We avoid multi-lock compaction schemes and deletion rebalancing to improve concurrency [35, 12]; this practice has been shown to have provably good worst-case properties [40]. To limit overhead from underfilled leaf nodes, a desired ratio of tree size to total key-value data can be selected. The tree can then be periodically rebuilt offline without sacrificing liveness, by checkpointing the tree, rebuilding it with insertions into an empty tree, then replaying a delta of operations performed since the checkpoint was recorded and swapping the live and offline trees.

We extend Sagiv's algorithm to server-side operations in our meganode structure.

(Server-side) search and caching: To search for a key-value entry, clients iteratively traverse the tree one meganode at a time by sending search requests to the appropriate servers, starting at the server containing the root node R (Figure 2). Each server uses Sagiv's algorithm to search within meganode(s) until it reaches a pointer that is not local to the machine. This remote pointer is returned to the client, which continues the search request at the pointer target's server. When a leaf node is reached, the server returns the key-value pair from its extents region to the client. To bootstrap the

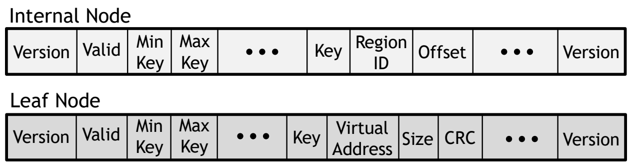

Figure 3: The structure of internal and leaf meganodes in Cell's B-link trees. Each node packs two matching versions, a minimum and maximum key, and zero or more pointers to other nodes or to extents memory in a block of a few kilobytes.

search, we ensure that a pointer to R is always stored at offset 0 in the region with the lowest ID across the cluster. We can speed up searches by having clients cache the depth and key range of meganodes close to the root because servers provide this metadata with traversal results. This type of caching is effective because updates in a B-tree such as ours occur exponentially infrequently in node height [40].

Deletion: To delete a key-value entry, a client orchestrates a server-side search for the key, then instructs the server to delete a leaf node entry according to Sagiv's algorithm. Nodes are not compacted to avoid multi-lock algorithms and maintain concurrency.

Insertion: To insert (or update) a key-value entry, a client performs a server-side search for the key, then instructs the server to insert a leaf node entry according to Sagiv's algorithm. As individual nodes fill and split (potentially propagating upwards) the meganode itself may become full, requiring that it split.

In principle, we could apply Sagiv's algorithm at the meganode level as well, but this would require locking the entire meganode for the duration of the split, blocking all other operations. Instead, we use a finer protocol inspired by Sagiv's algorithm that allows greater concurrency. It identifies a split key in a meganode X that divides it into two halves, X_{left} and X_{right}. X_{right} is locked for the duration of the split, but updates can continue in X_{left}. The server copies the nodes in X_{right} to a new local or remote meganode asynchronously[2]. Then, it locks X_{left} long enough to update the right-link pointers of X_{left} along the split boundary to point to the root of the new meganode. At this point, the meganode structure is restored as in Figure 2. Lastly, the server invalidates the old X_{right} by setting a boolean flag in each node, indicating that the nodes can be reused, and releases the lock on X_{right}'s key range.

A meganode should be split before it becomes too full, otherwise concurrent updates to X_{left} may fail if we run out of space. Note that client-side searches may occur throughout the meganode split process. Ensuring their correctness is subtle, as we discuss in Section 2.3.

[2]To balance network costs with the desire to balance the tree across all available servers, Cell favors remote meganodes when few local meganodes have been used, and local meganodes otherwise.

2.3 Client-Side Search and Caching

Cell organizes each meganode as a local B-link tree in order to enable client-side searches using RDMA reads. The search process is similar to the server-side equivalent, except that the client needs to iteratively fetch each B-tree node using an RDMA read, following child and right-link pointers, as it traverses a meganode. When the search terminates at a leaf node, the client attempts one additional RDMA read to fetch the actual key-value data from the server's extent region, or issues an insert or delete RPC to the server containing the leaf node.

A full-sized 64MB meganode built from 1KB-sized nodes contains a 5-level local B-link tree. Thus, RDMA search through a meganode takes up to 5 RTTs while server-side search requires only one. This is not as bad as it seems, because: (1) RDMA-enabled networks have very low RTTs, so the overall client-side search latency remains small despite the extra roundtrips. (2) The latency overhead pales in comparison to the queuing delay if the server CPU is bottlenecked at high load. To reduce search roundtrips and ameliorate hotspots at the root meganode, clients cache fetched nodes. Clients follow the same strategy as for server-side search: only cache nodes near the root, and if they follow a right-link pointer, they invalidate any information cached from the parent node.

Allowing client-side RDMA reads during server-side tree modifications introduces subtle concurrency challenges, as Sagiv's algorithm requires that individual nodes reads and writes appear atomic. This is easy to enforce among server-side operations using CPU synchronization primitives. However, no universal primitives exist to synchronize RDMA reads with the server's memory access. To ensure that RDMA reads are consistent, we use two techniques (see Figure 3):

- We store a version number at the start and end of each node. To update a node, the server increments both version numbers to the same odd value, performs the update, then increments both numbers to the same even value. Each step is followed by a memory barrier to flush the cache to main memory. If the RDMA read of a node is interleaved with the server's modification of that node, it will either see mismatched version numbers or the same odd value, indicating that the read must be retried. This method works because in practice RDMAs are performed by the NIC in increasing address order[3] and because individual nodes have fixed boundaries and fixed version number offsets.
- Key-value entries are variable size and have no fixed boundaries. We use the technique proposed in Pilaf [32] of storing a CRC over the key-value entry in

[3]To handle NICs that do not read in increasing address order, we need to adopt the solution of FaRM [7] of using a version number per cacheline.

the corresponding leaf node pointer. After performing an RDMA in the extent region, the client checks if the CRC of the data matches the CRC in the pointer; if not, the RDMA is retried. Like node modifications, each key-value write is also followed by a memory barrier.

2.4 Correctness

Cell dictionary operations (search, insert, and delete) are linearizable [16] so that both server-side and client-side searches will be correct. We refer readers to §4.2.1 of Mitchell's PhD thesis [33] for the proof.

Theorem 1. *Every Cell operation on small nodes maps a good state (in which all previously-inserted data that has not yet been deleted can be reached via a valid traversal) to another good state. Therefore, by the give-up theorem [41], it is linearizable.*

If meganodes never split, we can leverage Sagiv's proof of correctness [38]. However, it requires that node reads and writes be atomic, as guaranteed by the design in the previous section. Therefore, any search through a Cell store for x, if it terminates, will terminate in the correct place: the node $n \mid x \in keyset(n)$, where $keyset(n)$ is the set of keys that are stored in n. Since Cell satisfies the give-up theorem [41], and all Cell operations map good states to good states, Cell is linearizable.

The linearizability of operations during meganode splits is delicate, and requires an additional proof, also provided in Mitchell's thesis [33].

Theorem 2. *Insert, delete, and search operations remain linearizable during a meganode split. Insert and delete operations are blocked during the meganode split, and proceed correctly once the split is complete. Search operations can either continue or backtrack correctly during and after the split. Thus, Cell operations remain linearizable, by Theorem 1.*

The state stored in a node n allow multiple copies of a node to safely temporarily exist. Invalidating all but one copy of the node before allowing modifications to any copy of n removes any possible ambiguity in the set of keys present in Cell. The design for this is presented below. All Cell operations still map good states to good states as from Theorem 1. Thus, any search through a Cell store for x, if it terminates, will terminate in the correct node $n \mid x \in keyset(n)$, even during meganode splits.

Although the server can block its own write operations during a meganode split, it cannot block clients' RDMA reads, so we must ensure the latter remain correct. The problem occurs during node invalidation and reuse. By resetting the valid bit in each node in X_{right}, the server guarantees that concurrent client-side searches fail upon reading an invalid node. However, an invalid node might be reused immediately and inserted into an arbitrary location in X_{left}. A client-side search intending to traverse X_{right} might read this newly reincarnated node instead[4]. The min and max keys in each node allows a client to detect this case and restart the search.

2.5 Selectively Relaxed Locality

Traditional systems wisdom encourages maximizing locality between data and computation over that data to reduce expensive data copying over the network. RDMA greatly reduces the cost of moving data to computation, so Cell's hierarchical B-tree design allows for both server-side searches (over local data) and RDMA searches (over data copied from a Cell server). When servers are under low load and/or client resources are scarce, server-side searches achieve better overall resource efficiency. However, clients should switch to using RDMA searches when servers become overloaded and client CPUs are available. How should clients (and servers performing cross-server data structure modification) dynamically relax locality, i.e., decide which search method to use?

To answer this question, we model the system using basic queuing theory [14]. Specifically, we model each Cell server as consisting of two queues, one (Q_s) for processing server-side searches, the other (Q_r) for processing RDMA read operations. The service capacity (ops/sec) of Q_s is T_s, which is determined by the server's CPU capability, and the service capacity of Q_r is T_r, which is determined by the NIC's RDMA read capacity. We assume that Q_s and Q_r are independent of each other.

Let q_s and q_r represent the current lengths of the queues, respectively. Since our job sizes are fixed, the optimal strategy is to Join the Shortest Queue (JSQ) [13, 48]. This decision is made on a per meganode basis. More concretely, after normalizing queue length by each queue's service time, a client should join Q_s if $\frac{q_s}{T_s} < \frac{q_r}{T_r}$, and Q_r otherwise. We need to make another adjustment when applying JSQ: since each RDMA search involves m RDMA reads, the client should choose server-side search if the following inequality holds:

$$\frac{q_s}{T_s} < m \times \frac{q_r}{T_r} \tag{1}$$

Instead of directly measuring the queue lengths (q_s and q_r), which is difficult to do, we examine two more easily measurable quantities, l_s and l_r. l_s denotes the latency of the search if it is done as a server-side operation. It includes both the queuing delay and round-trip latency, i.e. $l_s = \frac{q_s}{T_s} + RTT$. l_r denotes the latency of an RDMA read during an RDMA search, i.e. $l_r = \frac{q_r}{T_r} + RTT$. Substituting $\frac{q_s}{T_s} = l_s - RTT$ and $\frac{q_r}{T_r} = l_r - RTT$ into inequality (1) gives us the final search choice strategy.

[4]Because nodes have fixed boundaries, we do not need to worry that a client read might read in the middle of a node.

To determine inequality 1, we need to estimate various terms. For a 64MB meganode with 1KB nodes, we initially set $m = 5$, the estimated height of each meganode in nodes; as we traverse meganodes, we adjust this estimate based on the average meganode height. We set RTT to be the lowest measured RDMA latency to the server. We approximate the current server-side search latency (l_s) and RDMA latency (l_r) by their past measured values. Over the time scale that the queue estimation computations are being performed, the rates of queue filling and draining do not change dramatically, so the average queue length also remains relatively stable. A client performing continuous operations may get new latency measurements (i.e., queue length proxy measurements) as often as every $10\mu s$ to $50\mu s$.

Additionally, we apply the following refinements to improve the performance of our locality selector:

- *Coping with transient network conditions:* We avoid modeling short-term transient network conditions with two improvements. First, we use a moving average to estimate l_s and l_r: clients keep a history of the most recent n (e.g. 100) samples for each server connection and calculate the averages. Second, we discard any outlier sample s if $|s - \mu| \geq K\sigma$, where μ and σ are the moving average and standard deviation, respectively, and K is a constant (e.g. 3). If we discard more samples than we keep in one moving period, we discard that connection's history of samples.
- *Improving freshness of estimates:* We perform randomized exploration to discover changes in the environment. With a small probability p (e.g., 1%), we choose the method estimated to be worse for a given search to see if conditions have changed. If a client has not performed any searches on a connection for long enough (e.g., 3 seconds), that connection's history is discarded.

The constants suggested above were experimentally determined to be effective on a range of InfiniBand NICs and under various network and CPU loads. We found that the performance of the selector was not very sensitive to changes in these values, as long as $K > 1$, p is small, and n covers timescales from milliseconds to one second.

2.6 Failure Recovery

Cell servers log all writes to B-tree nodes and key-value extents to per-region log files stored in reliable storage. The log storage should be replicated and accessible from the network, e.g. Amazon's Elastic Block Store (EBS) or HDFS. Our prototype implementation simply logs to servers' local disks. When a server S fails, the remaining servers split the responsibility of S and take over its memory regions in parallel by recovering meganodes and key-value extents from the corresponding logs of those regions. No remote pointers in the B-link tree need to be updated because they only store region IDs; the external (e.g., ZooKeeper-stored) map of region IDs to physical servers suffices to direct searches to the correct server.

Operation logging is largely straightforward, with one exception. During a meganode split, server S first creates and populates a new meganode before changing the right links of existing nodes to point to the new meganode. If S fails between these two steps, the new meganode is orphaned. To ensure orphans are properly deleted, servers log the start and completion of each meganode split and check for orphaned meganodes upon finding unfinished splits in the log. Node splits are handled similarly.

3 Implementation

We implemented Cell in ~18,000 lines of C++. Cell uses the `libibverbs` library, which allows user-space processes to use InfiniBand's RDMA and message-passing primitives using functions called *verbs*. We use Reliable Connection (RC) as the transport for both RDMA reads and SEND/RECV verbs. We use SEND/RECV verbs to create a simple RPC layer with messaging passing, used for client-server and server-server messages. Although client-side search requires that the client and server both have logic to traverse the tree, this code can be reused to limit the additional implementation effort.

Our server implementation is single-threaded; the polling thread also processes the Cell requests and performs server-server interactions for meganode splits. Like HERD [22], we run multiple server processes per machine in order to take advantage of multiple CPU cores. As suggested by FaRM's [7] findings on combining connections, we implemented a multi-threaded Cell client, and experimentally chose 3 threads per client process for most tests. To further increase parallelism, the client supports pipelined operations to avoid idling by keeping multiple key-value operations outstanding.

Client-side searches fetch 1KB nodes via RDMA to traverse a meganode. Server-side searches involve sending an RPC request to traverse a given meganode, and receiving the pointer to the meganode at the next meganode level along the path to that key's leaf node.

Clients cache B-link tree nodes to accelerate future traversals, maintaining an LRU cache of up to 128MB of 1KB nodes. We only cache nodes at least four node levels above the leaf node level to minimize churn and maximize hits. Symmetrically, each client maintains an LRU cache of up to 4K server-side traversal paths leading to the leaf-level meganodes, indexed by the key range covered by that meganode. We used in-band metadata to allow clients to verify the integrity of all RDMA reads. For the CRCs covering extents, we use 64-bit CRCs to make the probability of collisions vanishingly small. B-link tree nodes are protected by the previously discussed low and high version numbers.

Cell servers pre-allocate large pools of memory for B-link tree nodes and key-value extents. The extents pool is managed by our own version of SQLite's `mem5` memory manager. We support `hugetlbfs` backing for these large memory pools to reduce page table cache churn [7].

Node size and network amplification: We choose 1KB nodes for most of our evaluation. This number is not arbitrary: in our setup, 1KB RDMA fetches represent the point above which RDMA latency goes from being flat to growing linearly, and throughput switches from ops/sec-bound to bandwidth-bound.

RDMA-based traversals do incur a significant bandwidth amplification over server-side searches, on the order of $4 \cdot 1\text{KB}/64$ bytes $= 64\times$ with caching enabled. Using smaller nodes would shift the balance between bandwidth amplification and traversal time: for example, 512-byte nodes would add 4 levels (33%) to a 10^{15}-key tree while only enabling 16% more node fetches/sec per NIC. Client-side traversals would require 33% less bandwidth, but would take 12% longer.

4 Evaluation

We evaluated Cell's performance and scalability on the PRObE Nome cluster [1]. The highlights of the results:

- Cell adapts to server CPU and network resources across configurations. When the server CPU is more bottlenecked than its NIC (e.g. using a single core at the server), Cell achieves 439K searches/sec while server-side-only achieves 164K searches/sec. When the server is configured with 8 cores per server, Cell mostly uses server-side-processing, achieving 1.1 million ops/sec, 7% better than server-side-only and $3.7\times$ better than client-side-only.
- Cell handles load spikes that cause transient server CPU bottlenecks and increased queuing delay by increasing the ratio of client-side processing.
- Cell scales to 5.31 million search ops/sec using 16 Cell servers, with 2 cores each.
- Cell is effective for any mix of B-tree get and put operations, including those that cause online tree growth.

4.1 Experimental Setup

Hardware and configuration: Our experiments were performed on the PRObE Nome cluster. Each machine is equipped with 4 quad-core AMD processors and 32GB of memory, as well as a Mellanox MHGH28-XTC ConnectX EN DDR 20Gbps InfiniBand NIC and two Intel gigabit Ethernet adapters. Cell was run on top of CentOS 6.5 with the OFED 2.4 InfiniBand drivers.

For each experiment, we use distinct sets of server and client machines. Unless otherwise specified, we use:

- 4 server machines with 2 cores per server (by running 2 server processes per machine)
- the remaining machines in each experiment for clients
- 20M key-value pairs populated per server

Throughput at saturation and latency at a moderate (non-saturation) load are reported unless otherwise specified.

We enable `hugetlbfs` support on our server machines so that the RDMA-readable node and extents data can be placed in 1GB hugepages. Due to the complexity of modifying the InfiniBand drivers, we do not attempt to put connection state in hugepages, as our experiments indicate this would yield minimal impact on performance at scale due to other sources of InfiniBand latency [7].

We allow clients to consume up to 128MB of RAM to cache B-link tree nodes. To approximate performance with a much larger tree, we prevent the bottom four node levels of the tree from being cached, effectively limiting the cache to the top three levels in most of our tests.

Cell's throughput does not decrease when synchronous logging is enabled for key-value sizes below 500 bytes. For 100% put workloads that insert key-value pairs larger than 500 bytes, the I/O bandwidth of the SSD in each of our local cluster's servers becomes a bottleneck. As Nome's machines lack SSDs, we disable Cell's asynchronous logging in our experiments.

Workloads: To test search, get, and put (insert/update) operations, we generate random keys uniformly distributed from 8 to 64 characters, and values from 8 to 256 characters. We focus on evaluating the performance of search, as that is the dominant operation in any workload. For real-world benchmarks, we also utilize the Zipfian-distributed YCSB-A (50% puts, 50% gets) and YCSB-B (5% puts, 95% gets) workloads. All other tests use keys selected with uniform probability.

4.2 Microbenchmarks

Systems with relaxed locality are fast. Operating on a single server machine, Cell surpass the latency and throughput of schemes that maintain strict locality. This section evaluates Cell's search performance on a single meganode on a single CPU core without caching, using server-side search, client-side search, and selectively relaxed locality. Cell's performance on several cores per single machine is also measured.

Raw InfiniBand operations: We measure the throughput and latency of 1KB RDMA reads and 128-byte two-way Verb ping-pongs on 1 to 16 servers, utilizing 1 CPU core per machine (Table 1). Because very little per-message processing is performed on the servers, the CPU is not a bottleneck. We vary the client count to search the throughput-latency space for RDMA reads, Verb messaging, and simultaneous use of both. Because latency rises with no additional throughput past saturation, we report the throughput with the lowest latency within 5% of the maximum measured throughput.

Previous work has demonstrated higher throughput for RDMA and Verb messaging [22, 7]. FaRM reported $4\times$ 1KB RDMA read throughput on newer, $2\times$ throughput NICs [7]. We attempted to replicate HERD's published results using their benchmarks on the PRObE Susitna testbed, equipped with 40Gbps InfiniBand cards. We reached 16M ops/sec for one server with 4-byte RDMA reads accessing the *same* 4 bytes repeatedly within a 5MB region, compared with 20M ops/sec reported by the authors [22]. However, when we switched to 1KB RDMA reads spread through the entire 5MB region, the throughput dropped to 1.69M ops/sec, and further enlarging the RDMA-readable region to a more realistic 8GB dropped the throughput to 1.3M ops/sec. Notably, the tests throughout the remainder of this evaluation were run on older 20Gbps InfiniBand cards that are bottlenecked at a similar operation limit, 1.04M ops/sec over a 1GB area (or 1.44M ops/sec over a 64MB area).

Similarly, we tested HERD's Verb benchmarks. We achieved 12M ops/sec for 16-byte Verb messages exchanged over the Unreliable Connection (UC) transport, and 8.5M ops/sec for 128-byte messages over RC. In the HERD benchmark, each client only communicates with a single server. With one-to-one communication and a very small number of connections (10), we were able to achieve similar performance with our own benchmarks. Clients in a real-world distributed storage system need to engage in all-to-all communication with all servers, so our microbenchmarks (Table 1) report results with all-to-all communication with up to 100 clients.

We emphasize that selective locality can balance CPU and network, adapting to the available resources and providing better *relative* results than server-side or client-side processing alone. Some of our experimental conditions are more realistic than previous work, while we omit some difficult (but reasonable) optimizations that prior work uses. While our raw InfiniBand results differ in absolute numbers, Cell is able to balance the CPU and network resources at the level appropriate for a given environment, including those with better network performance and/or more CPU resources than our prototype.

Search throughput with a single meganode: To establish the baseline performance of Cell's selectively relaxed locality approach, we present a microbenchmark for traversals of a single meganode served by 1 server CPU core. We examine the performance of client-side only searches, server-side only searches, and Cell's locality selector, each with client-side caching disabled. Figure 4 demonstrates the throughput-latency relationship as we increase the number of client processes from 1 to 12, distributed across 4 client machines. With the server performing B-tree searches, 1 CPU core is no longer sufficient to saturate the NIC; the server-side search's peak throughput is 158K searches/sec, and the bottleneck is the server CPU. Client-side-only search achieves 305K search/sec; since 5 RDMA reads are required to traverse the 5-level meganode used and caching is disabled, this matches our raw InfiniBand performance of 1.44M ops/sec over a 64MB region. Combining the two methods yields 93% of the *aggregate* throughput, peaking at 432K searches/sec. For all except the 1-client (lowest-throughput) point on each line, the server's single CPU core is already saturated with the number of operations it can perform per second. RDMA provides Cell with extra throughput, and in this case lower latency, because the extra latency from multiple RDMA round trips is overshadowed by the queuing delay at the server for the server-side case.

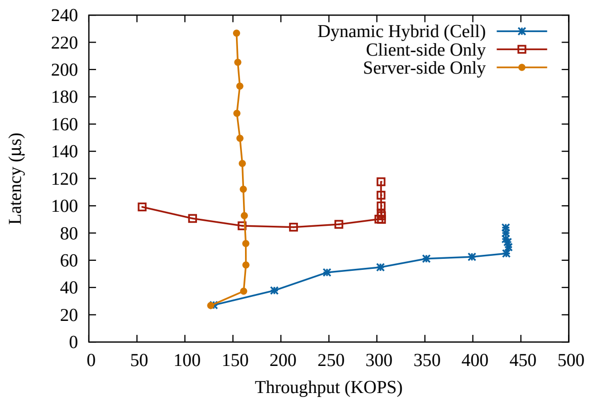

Figure 4: Throughput and latency for server-side search, client-side search, and Cell's locality selector. The experiments involve 1 server serving one meganode, utilizing one core. Client B-tree caching and operation pipelining are disabled.

To calibrate Cell's local in-memory B-tree implementation, we compare its performance and MassTree's [31]. Using key and value distributions matching Cell's and the `jemalloc` allocator, we measure MassTree's performance at 276K local B-tree searches/sec (using a single core), compared to Cell's 379K local searches/sec. This suggest Cell's local B-tree implementation is competitive. When adding the cost of network communication, Cell performs 158K server-side searches/sec. MassTree does not have InfiniBand support, nor does it implement a distributed B-tree. Thus, we do not compare with MassTree further.

4.3 Performance at Scale

Cell scales well across many servers. We vary the number of server machines from 1 to 16, using 2 CPU cores per server and enabling client caching. We scale the size of the B-tree to the number of servers, at 20M tuples per server, storing up to 320M tuples for 16 servers. We also use enough clients to saturate each set of Cell servers. Our biggest experiments consist of 64 machines (16 servers plus 48 clients) with 2560 client-server connections (80 3-threaded client processes to saturate the 32 server cores). Our largest B-tree is 2 meganodes tall and stores 320M tuples.

Throughput: Figure 5 shows the search throughput in logscale of Cell as well as the alternative of server-side processing only. It demonstrates that Cell displays near-linear throughput scaling over additional servers (satu-

	RDMA read		Verb messaging		Hybrid	
Servers	Throughput	Latency	Throughput	Latency	Throughput	Latency
1	1.04M ops	15.5μs	750K ops	21.4μs	1.60M ops	20.1μs
4	4.13M ops	15.5μs	2.96M ops	21.7μs	6.00M ops	21.8μs
8	8.77M ops	14.6μs	5.88M ops	21.5μs	10.58M ops	20.2μs
24	24.97M ops	15.5μs	18.15M ops	22.5μs	35.56M ops	22.0μs

Table 1: Microbenchmarks of throughput and latency at maximum throughput for 1KB RDMA reads over a 1GB area, 128-byte 2-way Verb messages, and both. All reported values are within 5% of the peak measured, at minimum latency.

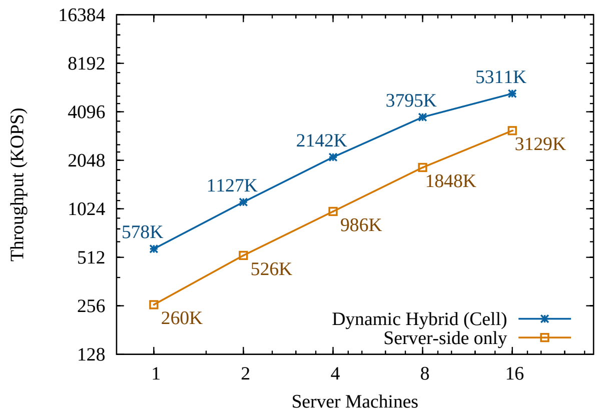

Figure 5: Throughput as the number of servers increases from 1 to 16. Each server uses 2 CPU cores. 4 client machines per server are used to saturate the servers.

rated with additional clients) until queue pair state overwhelms the NICs' onboard memory. Cell's throughput increases 9.2× from 1 server to 16 servers. Compared to the server-side only approach, Cell's hybrid approach achieves 70% higher throughput with 16 servers.

Latency: The B-tree remains at 2 meganode levels and 7 node levels as the number of servers grows from 4 to 16, so the overall search latency is stable. Below saturation, the median latency for both the server-side only approach and Cell is $\sim 30\mu s$.

4.4 Locality Selector: Balancing CPU and Network

Cell's locality selector effectively chooses the correct search method to use under arbitrary network and server load conditions. The selector is designed to minimize operation latency while rationing server CPU resources. We evaluate its performance by asking three questions:

1. How effectively does Cell use CPU resources?
2. How accurately does Cell estimate and balance search costs for a given environment?
3. When is the locality selector beneficial?

This section answers all three questions.

4.4.1 Varying Server CPU Resource

We measure the performance of Cell and server-side search from 1 to 8 server CPU cores on 1 server machine and 4 server machines. Table 2 compares Cell's locality selector and 100% server-side search on 4 servers, showing that Cell is able to consistently match or exceed server-side search's throughput. With 2 cores, Cell achieves 74% of server-side search's maximum throughput, for which the latter requires 7 cores. Cell's selective locality approach allows the system to dynamically balance server CPU, client CPU, and network bandwidth usage, so a Cell storage cluster can indeed economically satisfy transient peak usage with fewer CPU cores. In addition, in applications where server-side operations are more CPU-intensive and each CPU core can therefore complete fewer operations each second, Cell's advantage becomes even more pronounced.

Table 2 indicates that Cell's hybrid scheme can extract 2.13M searches per second from 2 CPU cores on each of 4 server machines, more than double the 991K server-side only searches at the same CPU count. Server-side searches are able to reach 2.90M searches per second using 7 cores per machine, close to the theoretical 3.0M ops/sec maximum Verb throughput our microbenchmarks suggest. This 3.5× resource expenditure yields only 35% higher throughput compared with the hybrid selector on 2 cores per machines, for example. The latency of hybrid operations remains consistently low even with Cell utilizing few CPU cores; with 2 cores and moderate load, searches average $28.5\mu s$, dropping to $26.4\mu s$ with 4 cores per server. Although modern servers typically have many more than 2 cores, even in a dedicated environment, cores no longer needed for storage logic can be devoted to CPU-intensive tasks data, including summarization, aggregation, analysis, cryptographic verification, and more. The server-side-only advantage in Table 2 for ≥ 7 cores is due to the fact that although only 1% of requests are performed client-side, with plentiful CPU resources the latency of client-side requests is significantly higher than server-side requests.

If CPU cores are plentiful, Cell has the same behavior as server-side processing and its performance is bounded by the InfiniBand IOPs. Using the Unreliable Datagram (UD) transport instead of the Reliable Connection (RC) transport can be used to avoid accumulating connection state and thus greatly increase messaging performance [24]. With UD, the higher messaging throughput can be saturated with additional server CPUs, if available, but RDMA is not available to provide load balancing. In a shared cluster with finite CPU resources handling a complex data structure like a distributed B-tree, the extra UD capacity would likely go to waste.

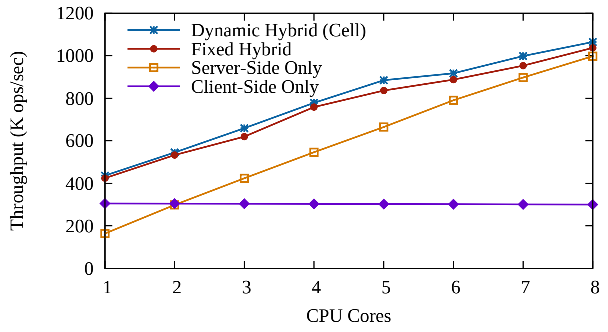

Figure 6: Throughput of 1 server running Cell on 1 to 8 CPU cores, using Cell's locality selector, a manually-tuned percentage of server-side operations, server-side only, and client-side only operations.

4.4.2 Dynamic Cost Estimation and Balancing

We find that under a constant server load, there is a per-client fixed ratio of server-side and client-side searches that produces maximal throughput at minimal latency. However, as illustrated in Table 2, these ratios shift dramatically as the available server resources change, and must be re-tuned for every change in the number of available cores per server, the number of servers, the number of clients, the workload, and the request rate. Cell correctly picks ratios that yield minimum median latency regardless of the server load; in fact, because the ideal ratio is slightly different for each client due to network topology and the position of its connection in the NIC's state, we observe Cell picking different ratios on each client that produce globally-optimized throughput and latency. Figure 6 shows that in most cases, especially when few resources are available, Cell meets or exceeds the throughput of the best hand-tuned fixed ratio, as it can continuously adjust for current load conditions.

Other environments: Tests on two other clusters with different NICs and CPUs yielded similar results. With similar 2-server-CPU experiments, Cell adapted with different ratios of client- to server-side search, using more client-side searches on Susitna's powerful NICs [1] and more server-side searches on our local cluster.

4.4.3 Load Spikes and Load Balancing

Cell's ability to maintain low latency in the face of transient server load demonstrates the value of dynamically selecting between RDMA and messaging-based traversal. In the absence of applicable traces from real-world systems at scale, we perform microbenchmarks of Cell's ability to rapidly adapt to load spikes and workload changes. Figure 7 compares the application-facing latency of Cell clients and clients that use only server-side search when the load on a cluster of 4 servers increases unexpectedly for 5 seconds. In these tests, application search requests arrive at each of 24 Cell clients every $75\mu s$. For 5 seconds, the load rises to $2.5\times$ as an additional 36 clients begin injecting searches at the same rate. Figure 7 shows that Cell is able to very rapidly switch the majority of its searches to client-side traversals, maintaining low median and 90th percentile latencies compared to server-side search. Cell's locality selector effectively manages server CPU resources to minimize latency in the face of long-term and short-term changes in server load and available resources.

Cores	Server-Side Only	Cell	Fixed Ratio
1	505K	1842K (30%)	1841K
2	991K	2142K (41%)	2129K
3	1447K	2456K (53%)	2366K
4	1831K	2573K (62%)	2513K
5	2145K	2687K (77%)	2593K
6	2478K	2845K (88%)	2768K
7	2901K	2901K (99%)	2776K
8	2543K	2812K (99%)	2571K

Table 2: Search throughput of 4 server machines utilizing 1 to 8 CPUs per server: 100% server-side searches, Cell (with the average of its dynamically-selected ratio of server-side searches), and a fixed percentage of server side searches using that average. On 8 or more CPUs, the amount of connection state necessary to connect servers to clients causes predictable degradation. Cell saturates with fewer clients, so this effect is less pronounced.

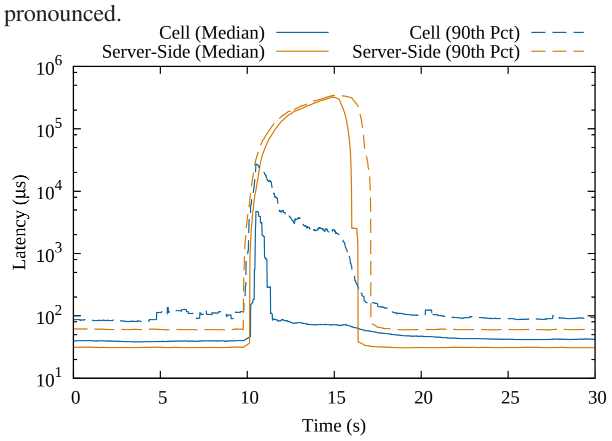

Figure 7: Request latency during 5-second transient load spike. These tests were performed with client pipelining disabled for simplicity. Note logarithmic y-axis.

4.5 Read-Write Concurrency

Like most sorted stores, Cell can perform search, get, put, delete, and range operations. We benchmark mixes of get and put operations to ensure that Cell retains its advantages beyond search operations.

Cell is designed to maintain low latency and to support concurrent read operations even when servers are modifying the tree state, for puts or for node or meganode splits. Notably, the locality selector optimizes for server utilization rather than allowing any single client to optimize for its own latency. Figure 8 traces the median latency of a group of 8 clients performing searches while a separate group of 16 clients idles, then performs bulk

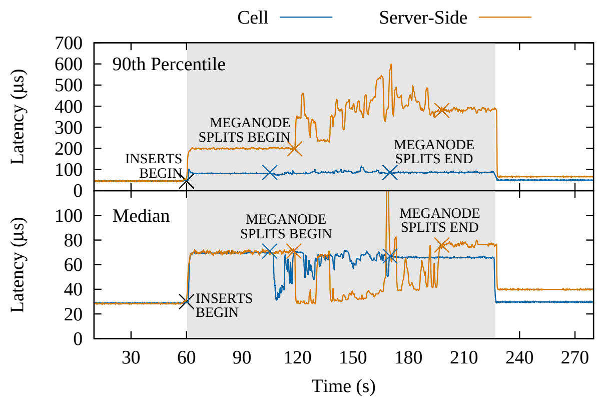

Figure 8: Latency of reads on 8 clients (24 threads) without and with 16 clients (48 threads) performing continuous insert operations on a cluster of 4 servers running 2 Cell threads each.

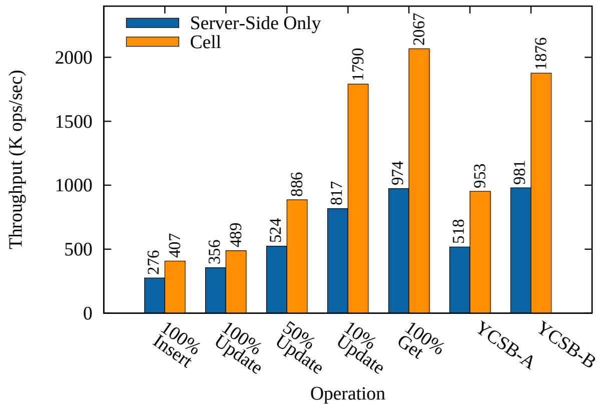

Figure 9: Throughput of pipelined mixed Get and Put workloads on 4 servers, 2 CPUs per server.

put operations that cause the tree structure to grow. With Cell, the search clients maintain concurrency by using more client-side operations, so a set of meganode splits completes faster. The median latency of Cell searches is thus slightly higher than the server-side equivalent, while the 90th percentile latency is far lower. Server-side search latency is impacted more significantly by write operations as the server's CPU capacity is shared between the two. With Cell we are able to dynamically shift the search operations to the client, incurring a slightly higher latency over server-side search, while using the saved server CPU cycles to execute node and meganode split operations with reduced latency.

Therefore, workloads that combine read and write operations can maintain lower latency (and higher throughput) with Cell than with server-side only operations. We test mixes of get (rather than search) and update operations from 100% get to 100% update, as well as 100% insert and two YCSB benchmarks. Figure 9 shows that Cell consistently outperforms server-side only operations across a range of workloads. We do not report the performance of range (range queries) because for queries that return a single key, range has the same performance as get. As the set of keys returned by range grows, the per-key range performance improves, as up to a full leaf-level node of keys can be returned by each RDMA fetch within the range operation.

5 Related Work

There are two general approaches to using RDMA in distributed systems. One is to use RDMA to improve the throughput of the message-passing substrate of a system, increasing the overall performance when the system is bottlenecked by its message-passing capability. We call this approach RDMA-optimized communication. The other approach is to use RDMA to bypass servers' CPUs to improve the performance of systems bottlenecked by that resource. We refer to this approach as systems with client-side processing. In this section, we discuss related projects exploring both approaches as well as work on distributed storage.

RDMA-optimized communication: The HPC community has exploited the performance advantage of RDMA extensively to improve the MPI communication substrate [28, 27, 42]. The systems community has recently begun to explore the use of RDMA in distributed systems. Most projects focus on using RDMA to improve the underlying message-passing substrate of systems such as in-memory key-value caches [22, 21, 20, 44, 23], HBase [17], Hadoop [29], PVFSpvfs and NFS [11]. For example, HERD has proposed using an RDMA write to send a request to a server and to have the server respond using an unreliable datagram (UD) message. FaRM [7, 8] also uses RDMA writes to implement a fast message passing primitive. The advantage of this approach is that the resulting solutions are generally applicable, because all distributed systems could use a high performance message-passing primitive for communication. However, when the server's CPU becomes the bottleneck instead of the network, this approach does not take full advantage of RDMA to relieve servers' CPU load.

Systems with client-side processing: Several recent systems exploit RDMA's CPU-bypassing capability. Pilaf [32] builds a distributed key-value store which uses client-side processing for hash table lookups. FaRM [7, 9] provides a distributed transactional in-memory chunk store which processes read-only transactions at the client. We could potentially layer Cell's design on top of FaRM, but this would require using distributed transactions to modify the B-tree [4, 43]. With Cell's approach, we do not need distributed transactions (Section 2). DrTM [47, 6] offers distributed transactions over RDMA by exploiting hardware transactional memory (HTM) support.

Cell is inspired by FaRM and Pilaf to offload the processing of read-only requests to the clients via RDMA reads. The main difference between Cell and prior work is that Cell explicitly tries to balance the server CPU and network bottleneck by carefully choosing when to prefer client-side over server-side processing.

Distributed B-trees: There are two high-level approaches for constructing a distributed B-tree. One builds a distributed tree out of lean (1-8KB) nodes [30, 4, 43]. Small nodes are crucial in the design of these systems, because the systems are all built on top of a distributed chunk/storage layer, and clients traverse the tree by fetching entire nodes over the network. The other approach, pioneered by Google's BigTable [5], builds a tree out of fat nodes (like meganodes; up to 200MB) [15]. This design reduces the number of different machines that each search needs to contact, but requires server-side processing to search within a meganode. Cell combines the two approaches.

Cell also handles concurrency differently than prior systems. Both Johnson and Colbrook [19] and Boxwood [30] implement distributed B-trees based on Sagiv's B-link tree. Johnson and Colbrook doubly-link the levels of the tree, merge nodes on deletion, and cache internal nodes consistently across servers, resulting in a complex scheme that requires distributed locks. Cell uses only local locks, similar to Boxwood. Furthermore, our caching of internal nodes is only advisory in that stale caches do not affect the correctness of the search. Aguilera et al. [4] implement a regular B-tree and handle concurrency requirements using distributed transactions, which are more conservative than necessary and are especially heavy-handed for read-only searches. Minuet [43] expands on Aguilera et al.[4], addressing some of the scalability bottlenecks and adding multiversioning and consistent snapshots. Some of their improvements emulate the B-link tree, yet they do not seem to benefit from the simplicity of Sagiv's single-locking scheme.

Other in-memory distributed storage: The high latency of disk-based storage has led to a large research effort behind in-memory storage. Memcached [10] and Redis [39] are popular open source distributed key-value stores. The RAMCloud project explores novel failure recovery mechanisms in an in-memory key-value store [36, 37], although it does not take particular advantage of RDMA. Masstree [31] and Silo [45] provide fast single-server in-memory B-tree implementations. MICA [26] presents a fast single-machine implementation of MassTree. These are not distributed and are not based on B-link trees.

6 Conclusion

RDMA opens up a new design space for distributed systems where clients can process some requests by fetching the server's state without involving its CPU. Mixing client-side and server-side processing allows a system to adapt to the available resources and current workload. Cell achieves up to 5.31M search ops/sec on 32 CPU cores across 16 servers, at an unsaturated latency of $\sim 30\mu s$. Cell saves up to 3 CPU cores per server per InfiniBand NIC, freeing resources for other CPU-intensive application logic, and maintains high throughput and low latency in the face of load spikes.

In looking towards Cell as the backend for real databases, we experimented with transactions and other database features. For example, RDMA can be used to quickly determine whether any item in a transaction's write set is currently locked by fetching many leaves across many servers simultaneously, and the status of locked keys can be re-checked with minimal resource consumption just by refetching that leaf.

From building and refining Cell, we learned lessons applicable to designing any distributed data store exploiting RDMA. Almost any RDMA-traversable distributed data structure that supports concurrent server-side writes can be built from CRC-protected variable-length data elements, version-protected fixed-length data elements, and a globally-known data structure root location. We learned that Pilaf's CRC approach works poorly for nodes in a structure with few roots, as each CRC update propagates to the root. Finally, an approach similar to our meganode split operation can make other types of distributed data structures' mutations friendly to concurrent RDMA access. Such mutations should replicate data, modify or rearrange data as necessary, then atomically (from the clients point of view) update links to old data to instead point to new data.

Acknowledgments

We would like to thank our shepherd, Peter Pietzuch, for his guidance and helpful suggestions. We appreciate our reviewers' insightful comments on this and earlier manuscripts that helped us hone this work.

Jinyang and Christopher were supported by NSF grant CNS-1409942.

References

[1] PRObE: testbeds for large-scale systems research. New Mexico Consortium and NSF and Carnegie Mellon University, `http://nmc-probe.org`.

[2] TritonSort: A balanced and energy-efficient large-scale sorting system. *ACM Transactions on Computer Systems 31*, 1 (2013).

[3] Extending high performance capabilities for Microsoft Azure, 2014.

[4] AGUILERA, M. K., GOLAB, W. M., AND SHAH, M. A. A practical scalable distributed B-tree. *Proc. VLDB Endow. 1*, 1 (2008), 598–609.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed

storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[6] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[7] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation* (2014).

[8] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 54–70.

[9] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of ACM Symposium on Operating Systems Principles* (2015).

[10] FITZPATRICK, B. Distributed caching with Memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[11] GIBSON, G., AND TANTISIRIROJ, W. Network File System (NFS) in high performance networks. Tech. rep., Carnegie Mellon University, 2008.

[12] GRAY, J., AND REUTER, A., Eds. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[13] HAIGHT, F. Two queues in parallel. *Biometrika 45*, 3 (1958).

[14] HARCHOL-BALTER, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[15] HBase. `http://hbase.apache.org/`.

[16] HERLIHY, M., AND WING, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[17] HUANG, J., OUYANG, X., JOSE, J., UR RAHMAN, M. W., WANG, H., LUO, M., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High-performance design of HBase with RDMA over InfiniBand.

[18] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference* (2010).

[19] JOHNSON, T., AND COLBROOK, A. A distributed data-balanced dictionary based on the B-link tre. Tech. Rep. MIT/LCS/TR-530, Massachusetts Institute of Technology, 1992.

[20] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for InfiniBand clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012).

[21] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011).

[22] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 295–306.

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference* (2016).

[24] KOOP, M. J. *High-Performance Multi-Transport MPI Design for Ultra-Scale InfiniBand Clusters*. PhD thesis, The Ohio State University, 2009.

[25] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 6*, 4 (1981), 650–670.

[26] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014), 36.

[27] LIU, J., JIANG, W., WYCKOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of MPICH2 over InfiniBand with RDMA support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (april 2004), p. 16.

[28] LIU, J., WU, J., KINI, S., BUNTINAS, D., YU, W., CHANDRASEKARAN, B., NORONHA, R., WYCKOFF, P., AND PANDA, D. MPI over InfiniBand: Early experiences. In *Ohio State University Technical Report* (2003).

[29] LU, X., ISLAM, N., WASI-UR RAHMAN, M., JOSE, J., SUBRAMONI, H., WANG, H., AND PANDA, D. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *International Conference on Parallel Processing (ICPP)* (2013).

[30] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *USENIX Symposium on Operating System Design and Implementation* (2004).

[31] MAO, Y., KOHLER, E., AND MORRIS, R. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), pp. 183–196.

[32] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference* (2013).

[33] MITCHELL, C. R. *Building Fast, CPU-Efficient Distributed Systems on Ultra-Low Latency, RDMA-Capable Networks*. PhD thesis, Courant Institute of Mathematical Sciences, New York, 8 2015.

[34] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcached at facebook. In *Proceedings of USENIX NSDI 2013* (2013).

[35] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I.

Berkeley DB. In *USENIX Annual, FREENIX Track* (1999), pp. 183–191.

[36] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[37] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 92–105.

[38] SAGIV, Y. Concurrent operations on B*-trees with overtaking. *J. Comput. Syst. Sci. 33*, 2 (1986), 275–296.

[39] SANFILIPPO, S., AND NOORDHUIS, P. Redis. `http://redis.io`.

[40] SEN, S., AND TARJAN, R. E. Deletion without rebalancing in multiway search trees. *ACM Trans. Database Syst. 39*, 1 (2014), 8:1–8:14.

[41] SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Trans. Database Syst. 13*, 1 (Mar. 1988), 53–90.

[42] SHIPMAN, G., WOODALL, T., GRAHAM, R., MACCABE, A., AND BRIDGES, P. InfiniBand scalability in Open MPI. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), pp. 10–pp.

[43] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet: A scalable distributed multiversion B-tree. *Proc. VLDB Endow. 5*, 9 (2012), 884–895.

[44] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost Memcached. In *Proceedings of USENIX Annual Technical Conference* (2012).

[45] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.

[46] VIENNE, J., CHEN, J., WASI-UR-RAHMAN, M., ISLAM, N., SUBRAMONI, H., AND PANDA, D. Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud computing systems. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on* (2012).

[47] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 87–104.

[48] WINSTON, W. Optimality of the shortest line discipline. *Journal of Applied Probability 14*, 1 (1977).

[49] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 523–536.