Building Fast, CPU-Efficient Distributed Systems on Ultra-Low Latency, RDMA-Capable Networks

by

Christopher R. Mitchell

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

September 2015

Professor Jinyang Li

DEDICATION

To Maria, Tori, and Pepper.

ABSTRACT

Modern datacenters utilize traditional Ethernet networks to connect hundreds or thousands of machines. Although inexpensive and ubiquitous, Ethernet imposes design constraints on datacenter-scale distributed storage systems that use traditional client-server architectures. Round-trip latency around $100\mu s$ means that minimizing per-operation round trips and maximizing data locality is vital to ensure low application latency. To maintain liveness, sufficient server CPU cores must be provisioned to satisfy peak load, or clients need to wait for additional server CPU resources to be spun up during load spikes.

Recent technological trends indicate that future datacenters will embrace interconnects with ultra-low latency, high bandwidth, and the ability to offload work from servers to clients. Interconnects like Infiniband found in traditional supercomputing clusters already provide these features, including $< 3\mu s$ latency, $40 - 80$ Gbps throughput, and Remote Direct Memory Access (RDMA), which allows access to another machine's memory without involving its CPU. Future datacenter-scale distributed storage systems will need to be designed specifically to exploit these features. This thesis explores what these features mean for large-scale in-memory distributed storage systems, and derives two key insights for building RDMA-aware distributed systems.

First, relaxing locality between data and computation is now practical: data can be copied from servers to clients for computation. Second, selectively relaxing data-computation locality makes it possible to optimally balance load between server and client CPUs to maintain low application latency. This thesis presents two in-memory distributed storage systems built around these two insights, Pilaf and Cell, that demonstrate effective use of ultra-low-latency, RDMA-capable interconnects.

Pilaf is a distributed in-memory key-value store that achieves high performance with low latency. Clients perform read operations, which commonly dominate key-value store workloads, directly from servers' memory via RDMA. By contrast, write operations are serviced by the server to simplify synchronizing memory accesses. Pilaf balances high performance with modest system complexity, disentangling read latency from server CPU load without forcing clients to complete difficult distributed RDMA-based write operations. Client RDMA reads can still conflict with concurrent server CPU writes, a problem Pilaf solves using *self-verifying* data structures that can detect read-write races without client-server coordination. Pilaf achieves low latency and high throughput while consuming few CPU resources.

0 ABSTRACT

Cell is a distributed in-memory B-tree store that can be traversed by client-side or server-side searches. Cell distributes a global B-tree of "fat" (64MB) nodes across machines, like BigTable. Within each fat node, Cell organizes keys as a local B-tree of small (1KB) nodes. It extracts the benefits of both server-side and client-side searches by switching between the two as server load changes. When servers are under unexceptional load, server-side searches are used to traverse entire fat nodes in single round trips. Clients maintain throughput in the face of load spikes by switching to RDMA-based searches over small nodes, traversing fat nodes in several round trips. The search process, whether client-side or server-side, is lock-free and correct with respect to concurrent tree modifications. Our evaluations show that Cell scales well and that the combination of server-side and client-side processing allows the system to respond nearly instantaneously to workload changes and load spikes while maintaining low latency.

RDMA and ultra-low latency make it possible to relax the data-computation locality necessary with traditional Ethernet networking. Through Pilaf and Cell, this thesis demonstrates that by combining RDMA and message passing to selectively relax locality, systems can achieve ultra-low latency and optimal load balancing with modest CPU resources.

ACKNOWLEDGMENTS

First and foremost, I owe a debt of gratitude my adviser, Jinyang Li, for her patient and tireless guidance throughout my doctoral program. Her expertise and unflagging passion for distributed systems helped fan the flames of my own interest and enthusiasm in the field. Thank you also to my co-adviser Lakshminarayanan Subramanian for his help and guidance through the years. Our floor administrator, Leslie Cerves, helped me navigate many of the harsher realities of academic paperwork and procedures, not to mention sharing my enthusiasm for a certain TV show about time travel. Most importantly, my mother Maria Mitchell supported and encouraged my education and the curiosity from an early age that led me eventually to this work.

I have been fortunate to have learned from and in turn guided many past and current members of Courant's Networking and Wide Area Systems group. Siddhartha Sen bestowed his knowledge and effort in collaborating on Cell. Russell Power guided me through my early publications, and provided valuable feedback on Pilaf. In chronological order, Yifeng Geng, Justin Lin, Kate Montgomery, and Lamont Nelson helped make Pilaf and Cell possible. Yang Zhang first suggested using CRCs to check for data inconsistency, and Frank Dabek suggested useful experiments to evaluate Pilaf's self-verifying data structure. Dennis Shasha guided the addition of serializability proofs to this thesis. Many members of the NeWS group, including Yang Zhang, Russell Power, Aditya Dhananjay, and Sunandan Chakraborty gave valuable feedback and support that helped improve this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

In the past decade, large-scale internet services hosted in datacenters have become ubiquitous in daily life. Distributed storage and computation systems make up much of the software in datacenters, while the hardware consists of commodity servers connected by a commodity interconnect. The CPU power and memory of these servers have steadily improved, but the network fabric connecting them largely remains inexpensive Ethernet. We believe that datacenter networks are on the brink of a revolution. The datacenters of tomorrow will be connected with networks offering bandwidth much higher than existing Ethernet infrastructure, latency orders of magnitude lower than Ethernet, and the ability to directly access other machines' memory in hardware. These features are not wishful thinking: the HPC interconnects used in supercomputing clusters over the past fifteen years have offered all of these.

In fact, HPC networking fabrics like Infiniband have already become popular options in commodity computing clusters [1, 9, 16]. Ethernet offers $\sim 100\mu s$ latency, up to 10Gbps bandwidth, and message passing. Infiniband, on the other hand, has $< 3\mu s$ round-trip latency, throughputs up to 80 Gbps, and Remote Direct Memory Access (RDMA), which allows a machine to directly read or write parts of a peer's memory without involving the remote machine's CPU. Historically, Infiniband has been expensive, but it recently has become an economical choice for datacenter networking: a 40 Gbps Mellanox Infiniband adapter costs ~$600, while 10 Gbps Ethernet cards range in price from ~$200 to $700. Surprisingly, low-latency Infiniband switches are now less expensive than their 10 Gbps Ethernet counterparts. Infiniband is not the only option; products are already being offered that implement kernel bypassing (to minimize latency) and RDMA over 10 Gbps Ethernet [73, 60].

With these new capabilities available in datacenter interconnects, it is important to understand how to build new distributed systems that take advantage of these features. Distributed systems used in datacenters fall into several categories, including distributed computation and distributed storage. This thesis focuses on distributed storage systems, which spread data to be read and written by clients across many machines. Distributed storage systems can be used as a cache layer for other systems, temporarily retaining frequently accessed data in memory [16], or as a persistent data store. Common classes of distributed storage systems include key-value stores, holding unordered collections of keys mapped to values [55], and sorted stores, used for applications requiring range queries and similar operations. Both types often

serve as the substrate for higher-level storage systems (e.g. BigTable [8], Spanner [11], Cassandra [37]) or computation frameworks (e.g. Piccolo [59]). This thesis investigates designing a distributed key-value store and a distributed sorted store. In order to build distributed stores that take advantage of low latency, high throughput, and RDMA, we must understand what new system architectures these capabilities facilitate.

1.1 Implications of Fast Networks on System Design

In order to understand what new possibilities HPC-like networks bring to system design, let's take a step back to see how traditional systems are structured. Traditional distributed storage systems spread data across the memory and/or disks of many servers, performing all computation necessary to maintain state and provide responses to client requests. Locality between computation and data is enforced by sending all computation to the servers' data: clients send requests via message passing, servers perform computations over local state, then the servers return responses via message passing. Sufficient server CPUs are provisioned to satisfy the expected load. When designed for Ethernet-based datacenter networking, which requires $\sim 100\mu s$ for each round trip, a central theme is minimizing the number of round trips. Maximizing locality minimizes the required round trips for each operation: the more related metadata (index) and/or data that is stored near each computation, the fewer round trips will be necessary to complete an operation.

Recent systems have explored Infiniband's potential to improve the performance of the traditional server-client model for distributed storage systems. A common approach [69, 31, 30] is to treat RDMA as a means for accelerating standard message passing. Clients send requests via message passing and include the local memory address of a receive buffer in the request. Servers compute and return responses by writing them directly into the client-provided receive buffer with an RDMA write. Another approach [32] is to exploit the high operation throughput of a subclass of Infiniband message-passing operations to achieve high throughput and low latency without departing from the traditional server-client architecture.

This thesis abandons the constraints of the traditional server-client model rather than trying to work within its limitations. We believe that Infiniband and similar HPC networking interconnects open up a more fundamental design space than accelerating server-side operations: performing some work on the clients, rather than the servers.

Insight 1. Relaxing locality between data and computation is practical.

Rather than using message passing to send computation to servers' data, RDMA can be used to directly read and write server state, performing computation at the clients and bypassing servers' CPUs. Clients can use RDMA reads to inspect servers' memory and traverse in-memory data structures, or RDMA read, write, and atomic operations to modify servers' memory and write new data. Because network operations are fast ($< 3\mu s$), many message-passing or RDMA round trips can be used to complete a single application operation while retaining extremely low operation latency. However, for stores with complex data structures under low CPU load, the additional network load required to copy data to clients overwhelms the benefits.

Insight 2. *Selectively* relaxing locality improves load balancing.

With complex data structures, communication costs may outweigh the benefits of moving data to computation when ample server CPU capacity is available. If server-side computation can complete an operation with many fewer round trips than client-side computation, the server-side approach will have lower latency, provided that servers are under moderate load. As server load rises, queuing effects increase the latency of server-side operations until client-side operations offer lower overall latency. Thus, selectively relaxed locality allows for novel load-balancing: when server CPUs are under high load, work can be shifted to clients' CPUs.

This thesis explores first if it's possible to have no locality for read operations. This hypothesis is the guiding principle behind the design of Pilaf, our distributed key-value store. Then, it examines what giving up locality achieves and costs, presenting a scheme where locality is selectively relaxed when it leads to better liveness or lower latency. This hybrid technique becomes the core principle of Cell, our distributed sorted store.

1.2 Pilaf and Cell

To test our conclusions about distributed storage systems on next-generation datacenter networks, we create prototypes of two important pieces of distributed systems infrastructure. Pilaf is a distributed in-memory key-value store for the many datacenter applications that require reading and writing isolated key-value pairs from a cache or store. Cell is a distributed in-memory sorted B-tree store for other applications

require a storage substrate that can maintain keys in a lexicographic order and perform range queries.

1.2.1 Relaxing Data-Computation Locality

Traditional distributed storage systems colocate the store's data and computation at the servers: clients issue read and write requests to servers, which compute and return responses. This thesis derives two key insights from the properties of HPC-like networks like Infiniband, and presents the implementation of the design suggested by each of these insights respectively in Pilaf and Cell. The data structure of a key-value store implemented as a distributed hash table is flat and simple: in its simplest iteration, a single location must be read to find a key, and a second read may be necessary to fetch the associated value. Realistic designs require searching through a small number of possible key locations. Key-value stores are thus a good candidate for Insight 1: with RDMA and ultra-low latency networks, systems can reduce locality constraints between data and computation. Pilaf decouples data and computation by delegating all read operations to clients' CPUs. Clients perform RDMA reads to directly read the servers' data structures. A B-tree store has a much more complicated hierarchical structure, requiring many reads to traverse from the root of the tree to a leaf. Therefore, Cell is a good candidate for our refined Insight 2: optimal performance is extracted from a distributed system on a next-generation datacenter network when data-computation locality is *selectively* relaxed. Cell uses ultra-low latency message-passing to perform traversal operations on servers, at the tree structure, when servers are under low load. When server CPU load rises to make this approach too expensive, clients switch to using RDMA reads to traverse the tree.

The costs of reducing locality for write operations are higher than for read operations. When a client fails to copy consistent data from a server to perform local computation, it must retry that read until it can proceed with additional computation. However, a client must perform some computation (for example, determining the new structure of a B-tree node) before each attempted (or retried) write to remote state. Therefore, we choose to avoid the high computation costs of conflicts with client-side writes, and instead follow traditional wisdom about data-computation locality for writes. Both Pilaf and Cell use server-side operations for put, delete, and similar write operations.

1.2.2 Pilaf: Practical and Efficient Client-Side Operations

Pilaf is a distributed in-memory key-value store that leverages RDMA to achieve high throughput with low CPU overhead. In Pilaf, we focus on Insight 1. Pilaf offloads *all* read operations, which dominate practical key-value store workloads [4], to clients. Clients find and fetch key-value pairs using a series of RDMA reads, bypassing the servers' CPUs. Because key-value stores are simple data structures – distributed hash tables with optional extents – values can be found in few small RDMA reads.

With clients simultaneously reading memory that servers may be writing, Pilaf reveals the most significant challenge to offloading work to clients' CPUs: memory access races. Short of incompletely supported hardware transactional memory, there is no way to synchronize RDMA memory accesses with CPU memory accesses. Pilaf's design restricts memory access races: clients might read inconsistent data while the server is concurrently modifying the same memory addresses, but no write-write conflicts are possible.

In Pilaf, we use *self-verifying* data structures to address read-write races between the server and clients. A self-verifying data structure consists of checksummed root data objects as well as pointers whose values include a checksum covering the referenced memory area. Starting from a set of root objects with known memory locations, clients are guaranteed to traverse a server's self-verifying data structure correctly, because the checksums can detect any inconsistencies that arise due to concurrent memory writes performed by the server. When a race is detected, clients simply retry the operation. CRC-protected self-verifying data structures are well-suited to small pieces of data that are modified infrequently, due to the cost of recomputing CRCs.

1.2.3 Cell: When to Use Client-Side Operations

Cell is a distributed in-memory B-tree store that combines client-side and server-side processing: clients can use RDMA or message passing to traverse the tree. Cell's design draws on Insight 2: data-computation locality is selectively relaxed for read operations by moving computation to the client when server CPU load raises the cost of server-side operations. To create a sorted store that can be traversed efficiently by client-side or server-side operations, Cell builds a hierarchical B-tree of B-trees. Globally, Cell is a distributed B-tree of "fat" nodes, each up to hundreds of megabytes in size, across server machines. Fat nodes enable efficient server-side processing; as the resulting global B-tree is shallow (often no more

than 3 levels), a B-tree search can be satisfied by contacting no more than three server machines, like a traditional distributed storage system [8]. However, to enable client-side search, Cell organizes the keys within each fat node as a local B-tree, consisting of small nodes each a few kilobytes in size. To search within a fat node, clients iteratively fetch the small nodes along the lookup path using RDMA reads.

One of the biggest technical challenges to this selective relaxation of locality is determining whether clients should perform an RMDA-based or server-side search. Ideally, when the server is not overloaded and traversing one unit of the structure via RDMA would require several round trips, clients should choose server-side searches. When the server is overloaded, some but not all clients should switch to performing client-side searches. If the structure consists of small objects that can be traversed in few round trips, client-side searches almost always outperform server-side searches. Cell allows clients to dynamically choose between server-side and client-side searches based on the estimated queuing delay of each search type, estimates based on the measured latencies of recent operations to the server. When the estimated queuing delay for server-side search exceeds that of the client-side alternative, the client picks the latter search type, which extracts more performance and allows for much-improved latency.

1.2.4 Implementation

We implemented both Pilaf and Cell on top of Infiniband, a popular HPC network interconnect offering RDMA and ultra-low latency. Our experiments on our local 10-machine cluster and a subset of the 256-machine PRObE Nome cluster demonstrate that Pilaf and Cell achieve high performance, good scalability, and efficient CPU utilization. In a workload consisting of 90% gets and 10% puts, Pilaf achieves 1.3 million ops/sec while utilizing only a single server CPU core, compared to 55K for Memcached and 59K for Redis. Pilaf's self-verifying data structures correctly detect all memory access races.

With 16 server machines each consuming two CPU cores, Cell achieves 3.13 million ops/sec doing pure server-side searches and 5.57 million ops/sec combining both server-side and client-side searches. Furthermore, Cell clients make good dynamic decisions on when to relax locality, consistently matching or exceeding the best manually tuned fixed percentages of client-side and server-side searches. They respond nearly instantaneously to high server load, maintaining low operation latency by quickly shifting to client-side searches.

1.3 Comparisons with Other RDMA-Aware Systems

Related datacenter-scale systems have explored distributed stores with RDMA and low-latency message-passing operations; Chapter 7 provides a complete discussion of these and other related systems. Three Memcached-like key-value stores [69, 31, 30] treat RDMA as a means for accelerating standard message passing. Three other systems [16, 32, 7] are much closer to Pilaf, using RDMA and/or low-latency messaging to take full advantage of Infiniband's advantages. We know of no RDMA-aware distributed sorted stores similar to Cell.

The three Memcached-like systems use RDMA to accelerate message passing, rather than as a means of accessing server state. For example, each client sends a get request to the server, which retrieves the corresponding key-value pair and directly stores it into the client's memory using an RDMA write. By contrast, Pilaf and Cell clients can process get requests without involving the server process at all, resulting in zero server CPU overhead. To the best of our knowledge, Pilaf was the first system design where clients can completely bypass the server's CPU to complete read requests.

FaRM [16], HERD [32], and Nessie [7] each explore regions of the solution space that Pilaf and Cell employ. FaRM [16] is an unordered key-value stores where the get operation (i.e. hash table lookup) is performed by clients using RDMA. Compared with B-tree traversals, hash table lookups are more efficient when processed at the client, requiring at most 2 RDMA reads. Nevertheless, HERD [32] has demonstrated that by provisioning sufficiently many server CPU cores, the throughput of server-side lookup processing can saturate the network card and exceed the performance of client-side processing. Nessie [7] explores a path that each of the other systems (including Pilaf and Cell) avoid as impractical: using client-driven RDMA writes to bypass the server's CPU for both read and write operations. It takes Insight 1 to the extreme of severing data-computation locality for all operations.

1.4 Contributions

Exploring RDMA-capable, ultra-low latency network fabrics for datacenter-scale distributed systems leads to several key conclusions about future systems design choices. We believe the architectures of Pilaf and Cell exemplify choices that future systems architects will need to make in the near future. In particular, this thesis adds the following important contributions to the state of the art:

CPU-efficient systems relax data-computation locality: In Section 1.2.1, we posited that distributed storage systems can make optimal use of new datacenter networking features by relaxing the enforcement of locality between data and computation. The simpler the structure of in-memory state, the more advantageous it becomes to shift a large portion of the client operations to RDMA. To reduce wasted computation in the face of conflicting operations, locality is only relaxed for read operations; systems maintain traditional locality for write operations.

Self-verifying objects make RDMA-based systems practical: Pilaf and Cell present three techniques for building data structures out of self-verifying objects. For small, fixed-sized objects, a checksum over the object's data is concatenated to the object to verify that the object is in a consistent, readable state. For pointers to variable-sized data, Pilaf and Cell both use checksums computed over the data and stored in fixed-sized self-verified objects, ensuring that both the pointer and the contents of the variable-sized data are consistent. For large fixed-sized objects, Cell uses version-bounded objects and memory barriers, once again ensuring that peers will repeat RDMA fetches of objects they cannot verify to be internally consistent. Reliable self-verifying objects used to build data structures that can be correctly traversed in a lock-free, read-only, unsynchronized fashion are a key component of RDMA-aware distributed systems.

Selective locality relaxation yields better load balancing: While the Pilaf design suggested by Insight 1 provides high throughput and low latency for distributed stores with simple structures, more complex data structures require a more nuanced approach. When a server-side read operation uses significantly fewer network round trips than a client-side version of the same operation, and server CPU capacity is available, data-computation locality should be maintained to minimize latency. As server CPU load increases, a threshold will be reached where client-side operations have lower latency despite additional round trips. Cell demonstrates successful use of heuristics to estimate server-side and client-side operation latency and select the instantaneously correct option.

Locality-relaxation techniques work at scale: Pilaf and Cell achieve high performance, good scalability, and efficient CPU utilization by unilaterally or selectively relaxing data-computation locality for read operations. Pilaf moves computation for all read operations to the client, and in a workload consisting of 90% gets and 10% puts, achieves 1.3 million ops/sec on a single server CPU core. Cell selectively moves computation to the client based on predicted server-side and client-side latency. With 16 server machines

each consuming two CPU cores, Cell achieves 3.13 million ops/sec doing pure server-side searches and 5.57 million ops/sec combining both server-side and client-side searches.

The chapters in this thesis will ground these contributions, beginning with an introduction to the opportunities HPC networking fabrics present and moving through the design, implementation, and evaluation of Pilaf and Cell. Chapter 2 presents the functionality and performance of current HPC Infiniband networking. The opportunities and challenges that RDMA-capable networks offer contrasted with the architecture of traditional Ethernet-based distributed systems lead us to the designs of Pilaf in Chapter 3 and of Cell in Chapter 4. The implementation-specific details that make it possible to turn these designs into a pair of working systems are presented in Chapter 5. Pilaf and Cell's performance and design choices are thoroughly evaluated in Chapter 6. Chapter 7 explores related systems, and this thesis is capped off with a conclusion of our results and contributions in Chapter 8. Datacenters are poised to adopt HPC-like networks, and selectively relaxed locality is key to designing next-generation distributed systems.

2
OPPORTUNITIES AND CHALLENGES

The arrival of high-performance networking in the datacenter will fundamentally change systems designs that have heretofore been dictated by the limitations of Ethernet. Ethernet is by far the best-known and most widely used networking system in consumer, enterprise, and even high-performance computing. From aging 100 Mbps to modern 1 Gbps and 10 Gbps links, to bonded Ethernet that combines the bandwidth of several Ethernet links, it is ubiquitous. Ethernet is supported by decades of hardware and software, and its strengths and limitations are well-understood by the systems and networking communities. Ethernet hardware is inexpensive, even for enterprise-grade devices. However, Ethernet's very suitability to a wide variety of networking scenarios makes it a suboptimal choice for high-performance computing, and HPC cluster architects have thus sought other options. Datacenters supporting distributed systems continue to largely use traditional Ethernet, however, forcing systems designs to account for the high round-trip latency of Ethernet networks.

We believe that tomorrow's datacenters will use HPC-like networks to interconnect racks of servers, and the systems built for datacenters will have to be able to take advantage of newly available networking features. This chapter gives an overview of RDMA and ultra-low latency networking and discusses how they might impact the design of distributed systems. Our discussion of the performance implications is based on Infiniband, a popular HPC interconnect.

Manufactured by Intel and Mellanox, Infiniband NICs provide 20, 40, or 80 Gbps of bandwidth in each direction. These NICs are also called *Host Channel Adapters* (HCAs); this thesis uses NIC and HCA interchangeably to refer to Infiniband cards. Infiniband HCAs are connected though Infiniband switches, which offer extremely low latency ($\ll 1\mu s$), wire-speed throughput, and usually full bisection bandwidth. Applications running on top of Infiniband have several communication options:

IP over Infiniband (IPoIB) emulates Ethernet over Infiniband. As with normal Ethernet, the kernel processes packets and copies data to application memory. IPoIB allows existing socket-based applications to run on Infiniband with no modification.

Send/Recv Verbs provide user-level message exchange: these message Verbs (heretofore called Verb messages) pass directly between userspace applications and the network adapter, bypassing the

kernel. Send/Recv Verbs are commonly referred to as two-sided operations since each Send operation requires a matching Recv operation at the remote process. Unlike IPoIB, applications must be rewritten to use the Verbs API.

RDMA allows full remote CPU bypass by letting one machine directly read or write the memory of another machine without involving the remote CPU. Unlike Send/Recv Verbs, RDMA operations are one-sided, since an RDMA operation can complete without any involvement of the remote process or CPU. RDMA is technically a type of Verbs. This thesis uses the term RDMA specifically to refer to RDMA Verbs and the phrase Verb messages to refer to Send/Recv Verbs.

Infiniband is not the only network to support RDMA and user-level networking. Similar features have recently been made available in 10 Gbps Ethernet environments. For example, both Myricom and Solarflare offer 10GE adapters that support kernel bypass, and Intel offers 10GE iWARP adapters capable of RDMA over Ethernet. Although it remains unclear which specific hardware proposal will dominate the datacenter market, future datacenter networks can be realistically expected to support some form of CPU bypassing.

2.1 Future Datacenter Networking Oracle: Infiniband

Infiniband is a useful proxy for future HPC-like datacenter interconnects: it offers all the features we expect in these networks. To map a hypothetical datacenter interconnect onto Infiniband, it is useful to understand the transport modes and operations that Infiniband offers. This section begins with the progression of a single message-passing operation in Reliable Connection (RC) mode, a TCP-like transport mode, in Section 2.1.1. Applications specifically rewritten for the Verbs API have several ultra-low latency message-passing and RDMA operations available to them; Section 2.1.2 describes these operations in more detail. Like IP over Ethernet, Infiniband offers several connectionful and connectionless protocols (also called *transport modes* [50]) that we believe represent adequate coverage of the possible protocols future interconnects will offer. Section 2.1.3 describes these protocols and the operations and guarantees provided by each. Finally, Section 2.1.4 adds information about additional features provided by Infiniband that make communication faster.

2.1.1 Infiniband Internals and Terminology

In exchange for extremely low latency and finer-grained control over networking, some of the burden of interfacing with the NIC shifts from the kernel to libraries and to the Infiniband-enabled application itself. To demonstrate how Infiniband operations work and to clarify some of the extensive related terminology, this section will trace the progression of a single message-passing send operation. The timing diagram for this operation is shown in Figure 2.1. Note that time proceeds from top to bottom, and that two peers connected over the Reliable Connection (RC) transport mode are shown. The connection has already been established, so each peer already has a *queue pair* (QP) exclusively used for that connection. In addition, this operation is shown as *signalled*, meaning that the sending application receives a notification when the send completes.

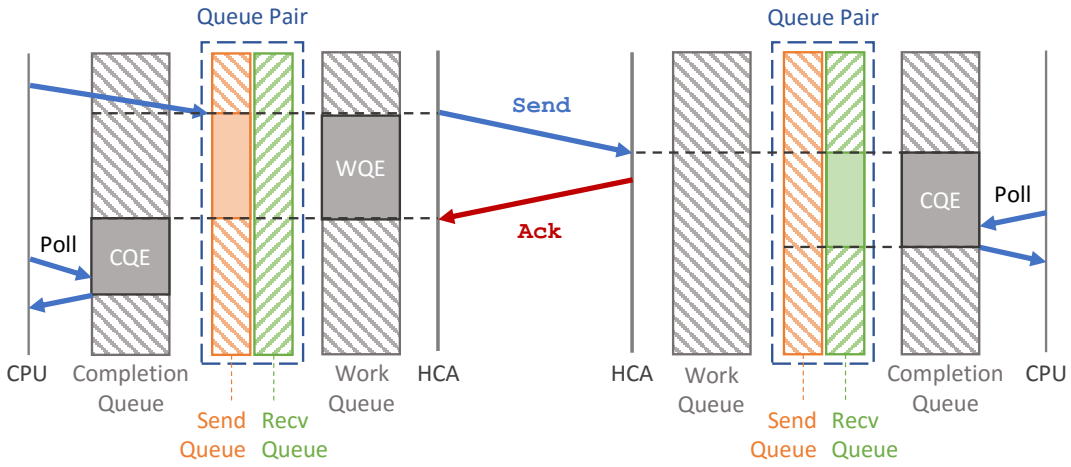


Figure 2.1: The progression of a single Infiniband send operation from one peer to another peer over a connection in reliable (RC) mode. The message is not sent with inline data, so the send buffer (and send request) are not freed until an acknowledgment is received from the peer.

The operation begins when the user application on the local (left-hand) peer instructs the userspace library to post a send operation to the HCA. The library translates the send request (a *scatter-gather entry* or SGE) into a hardware-specific request and gives it to the HCA. The HCA puts the send request into the *send queue* for that connection's queue pair, stored in its on-board memory [57], then sends the message out over the wire. Upon setting up the connection, both peers had posted a number of receive buffers to the HCA, ready to be used for incoming messages. The remote (right-hand) peer stores the

incoming message into a receive buffer and puts it into the queue pair's receive queue. It also places a *Completion Queue Entry* (CQE) into the *Completion Queue* (CQ), which can be shared between one or more connections. The remote application can either receive a notification via a pre-registered callback, or as in this diagram, periodically poll the completion queue for new CQEs. When it does so, the CQE is removed; the application is responsible for notifying the HCA when the receive buffer can be reused.

After writing the incoming message successfully into a receive buffer, the remote HCA sends an acknowledgment (ACK) message back over the wire. The local HCA removes the WQE for this send upon receiving the ACK, and inserts a new CQE into the sender's CQ. When the local application polls the CQ, it will receive this CQE, and can use it as a hint to (for example) mark the send buffer for this message as reusable.

2.1.2 Operations

Infiniband offers standard message-passing send and receive operations, as well as the ability to directly read and write a remote machine's memory. Qualitatively, use of Infiniband operations is more complex than Ethernet sockets: applications must set up buffers for sending/writing and receiving/reading data manually, applications control when buffers are considered reusable, and applications control the depth and draining of incoming (CQ) and outgoing (WQ) event queues. With this complexity comes much finer-grained control over networking, including zero-copy operations and extremely low latency.

The following operations are available to all applications that use the Verb API. Figure 2.2 diagrams the intra- and inter-machine communication performed for each operation. Depending on which transport mode is in use (see next section), only a subset of these operations may be used between two specific machines.

RDMA Read: Applications can read an arbitrary section of a peer's memory into a local buffer without involving the remote CPU. RDMA reads are only available in the reliable connection transport mode (see Table 2.1). In order to perform an RDMA read on a remote machine, the remote machine must explicitly register a region of memory with its HCA, and the local machine must acquire a key for the given memory region. The only current hardware synchronization primitives available to coordinate memory reads between the remote machine's CPU and HCA are the hardware transactional memory features available in the latest Intel processors [7]. However, we do not have access to a

cluster with these features, and in its current state only CPUs can initiate transactions. RDMA reads and writes performed by a single HCA *are* guaranteed to follow some global ordering, however.

RDMA Write: Similarly, applications can overwrite data in an arbitrary pre-registered area of a peer's memory. As with RDMA reads, the remote host must register the memory region, and the local host must fetch a key from the remote host to be able to write that memory region. Symmetrically, no synchronization primitives between RDMA writes and CPU read/writes exist, but RDMA operations performed by a single HCA are linearizable. RDMA writes are only available over the reliable connection and unreliable connection transport modes.

RDMA Atomic Operations: Some Infiniband HCAs offer atomic operations, which combine a read and a write operation into a single atomic unit. RDMA atomic operations are strictly serializable with respect to all other RDMA read, write, and atomic operations, but cannot be coordinated with CPU accesses to the same memory. The two available atomic operations are fetch and add, which adds a specified value to a value in RAM and returns the original value (*i.e.*, a pre-sum operator), and compare and swap, which can be used to build more complex synchronization primitives like mutices.

Verb Send/Receive: Infiniband's send and receive operations are similar to traditional Ethernet message-passing. Applications use the message-passing operations via the same userspace library as for RDMA, which requires them to create send and receive buffers and register them with the HCA. Applications track which send buffers are currently in use, and receive notifications from the HCA when a send buffer can be reused. Applications also receive notifications when incoming messages are delivered into receive buffers, and must re-post those receive buffers when they can be reused for new incoming messages.

One additional variation exists. Applications can request that an immediate address-length value be transmitted along with an RDMA write request. If this RDMA Write with Immediate operation is used, then the remote host is notified that the write has been performed and receives the immediate value, closely resembling the effects of the send operation [50].

Table 2.1 details which of these operations are available in each of Infiniband's various transport modes. To understand this table, knowledge of how each transport mode works is helpful.

Feature	Unreliable Datagram (UD)	Unreliable Connection (UC)	Reliable Connection (RC)
Send	•	•	•
Receive	•	•	•
RDMA Write		•	•
RDMA Read			•
RDMA Atomic			•
Guaranteed Delivery			•
In-Order Delivery			•
Multicast	•		

Table 2.1: Infiniband transport modes, plus the features each provides [50]. Not all HCAs and APIs support all transport modes (especially Reliable Datagram mode, which is omitted from this table). Some HCAs do not support RDMA atomic operations.

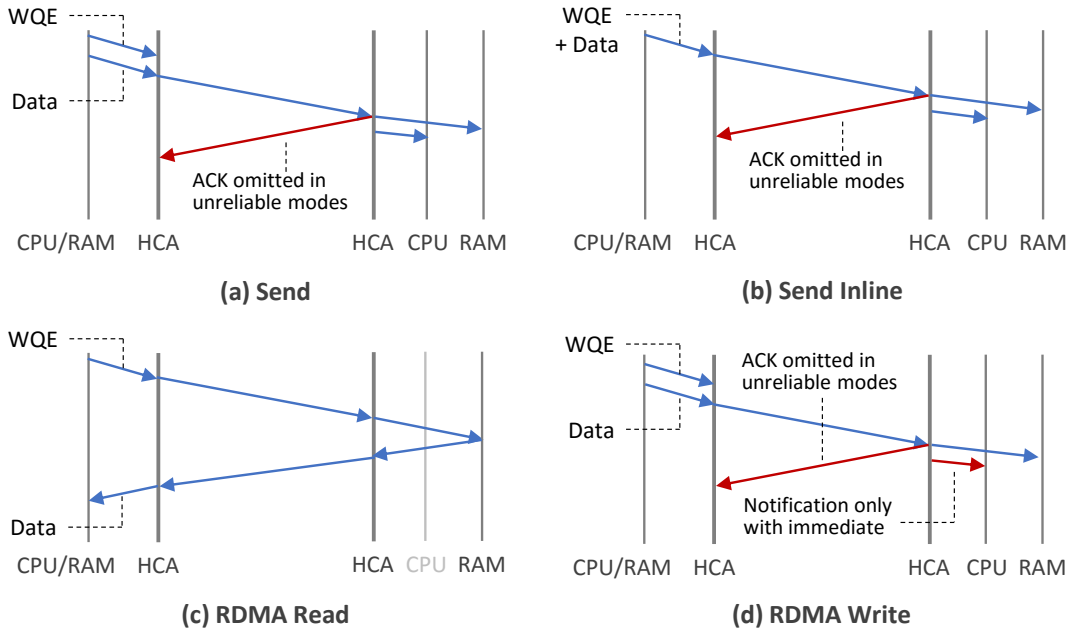


Figure 2.2: Infiniband operations in actions [32, 50]. Each operation is initiated by the machine on the left side of the diagram. Time increases down each timing diagram. Note that RDMA writes without immediate data and RDMA reads do not involve the remote CPU.

2.1.3 Infiniband Transport Modes

Like IP over Ethernet, Infiniband offers several transport modes, or protocols. Pilaf and Cell use the Reliable Connection (RC) mode, which is very similar to TCP. Messages have guaranteed delivery, are delivered in-order, and the HCA is responsible for retrying transmission until each message is sent. Infiniband

also offers a UDP-like protocol, Unreliable Datagram (UD) mode, and a hybrid of the two, Unreliable Connection (UC), which provides faster, lighter-weight communication than RC mode. Table 2.1 lists the features and operations available in each transport mode; each offers a different balance of speed and features.

Infiniband has two reliable transport modes that guarantee message delivery. Reliable Connection or RC mode is most similar to TCP. Messages are guaranteed to be delivered, and are sent in-order. Connections must be established between two peers before messages can be sent; the connection is formed between a queue pair (QP) on each peer that is not connected to any other QP. There is also a Reliable Datagram (RD) mode is not supported by the Mellanox Verbs API, but offers connectionless communication with guaranteed delivery.

Infiniband also offers two unreliable transport modes that do not guarantee message delivery, but are faster and more efficient. Specifically, because message delivery is not reliable, the overhead of sending an acknowledgment on receipt of each message is avoided. Unreliable Connection or UC mode requires a connection like the Reliable Connection mode, but does not guarantee message delivery. In practice, Infiniband's flow control (see next section) makes it very uncommon for messages to be lost unless a hardware error occurs [32]. Unreliable Datagram (UD) mode offers properties similar to UDP. Messages delivery is not guaranteed and no message ordering is maintained. No connections are used; instead, any QP can communicate with any other QP. In addition, like UDP, UD offers one-to-many multicasting.

2.1.4 Additional Features

Infiniband as expressed by the Verbs API offers several additional features that make the operations discussed in this section more useful.

- Applications can choose to interact with the HCA via a **callback or polling**. Pilaf and Cell continually poll for incoming Completion Queue Events (CQEs) that indicate a send, receive, or RDMA read operation has completed.

- For small messages, applications can **inline the data** to be sent with the Work Queue Entry (WQE). The data to be sent is given to the HCA in the same PCIe transfer as the destination and other metadata, saving a transfer and reducing latency [46]. In addition, the send buffer can be reused immediately. Pilaf and Cell utilize inline sends for messages no larger than 400 bytes.

- Applications can specify an address-length (32- or 64-bit) **immediate value** to be included with send and RDMA write operations. This value is stored in the message metadata rather than the message data. Immediate data can also be attached to RDMA read operations: this value is never sent to the peer, but is instead returned to the application with the data read from the remote host. Cell uses the immediate value to identify which entity (thread, job, or pipeline worker) triggered that RDMA read operation and return the result to the correct entity.

- While the unreliable transport modes (UC and UD) do not guarantee delivery, Infiniband's effective credit-based **flow control** makes message loss unlikely without hardware failure. Some systems have used UC mode instead of RC mode to exploit this practical reliability for higher performance with minimal cost [32]. Pilaf and Cell require RDMA read operations, and thus use the RC mode. In Infiniband's flow control scheme, each receiver computes a credit limit from the data received plus the available buffer space for new data [51]. The receiver periodically sends a *credit packet* to the attached transmitter (*i.e.*, its peer) indicating its credit limit. As long as the transmitter's total data sent on that link plus the size of the next packet to send does not exceed the peer's credit limit, it can continue transmitting. Because lost packets can cause the sender's and receiver's accounting of the total data transmitted across the wire to fall out of synchronization, credit packets are also periodically sent from the transmitter to the receiver.

- Applications can specify whether work request (WR) operations should be **signaled**: whether completed send operations or the send half of an RDMA operation should generate a completion queue event. Applications can reduce latency by choosing to issue unsignaled operations, which pair particularly well with inline sends. However, Infiniband does not allow all operations to be unsignaled: HCAs require that every N sends, at least one be signaled (for Mellanox cards, $N \approx 128$ [32]).

- **Infiniband switches** are fast [34] and inexpensive [54]. They offer port-to-port switching times of hundreds of nanoseconds ($\sim 200ns$ [5]), and full bisection bandwidth.

These features, together with the transport modes and operations described herein, combine to make Infiniband an efficient ultra-low-latency, high-speed transport suitable to represent future datacenter networks. To further demonstrate Infiniband's suitability to our exploration of future datacenter systems architecture, we explore the performance of Infiniband RDMA and message-passing operations.

2.2 Performance Benefits

How fast and efficient is Infiniband? How does the performance of RDMA and verb operations compare to alternatives such as traditional kernel-based TCP/IP transport? We answer these questions by benchmarking the various Infiniband communication options.

2.2.1 Single-Server Performance

We start by measuring Infiniband's performance on a single server with a single HCA. These experiments were run on a small cluster of machines equipped with Mellanox ConnectX VPI 20Gbps (DDR) Infiniband cards. For RDMA experiments, each client node performs RDMA reads on the server. For Verb message experiments, each client node issues a request (as a Verb message in reliable connection mode) to which the server responds immediately with a reply. The IPoIB and Ethernet experiments are similar except that we use TCP/IP for exchanging requests and replies. We vary the size of the RDMA read or the request message while fixing the reply size at 10 bytes.

Figure 2.3 shows the round trip latencies of different communication methods. For small operations ($<$ 1024 bytes), a Verb message exchange takes less than 10μs, while the RTT of IPoIB or Ethernet is over 60 μs. Our Infiniband switch imposes a lower delay than our Ethernet switch, but the IPoIB latency is similar to that of Ethernet, suggesting that packet processing through the kernel adds significant latency. RDMA achieves the lowest RTT ($\sim 3\mu$s), half that of Verb messages. This is because the request/reply pattern of traditional messaging involves two underlying Verbs exchanges. By contrast, an RDMA operation involves only one underlying exchange, thereby reducing the latency by up to half.

	Throughput (M ops/sec)		
Transport	16-byte	1024-byte	4096-byte
RDMA	2.449	1.496	0.472
Verbs Message	0.668	0.668	0.464
IPoIB	0.126	0.122	0.028
Ethernet (1Gbps)	0.120	0.068	0.029

Table 2.2: Throughput (in million operations/sec) for 16 byte, 1Kbyte and 4Kbyte operations.

Figure 2.4 shows the throughput (in Kbps) achieved by the server. Since different communication methods incur varying CPU overhead, we limit the server's CPU consumption to a single core (AMD Opteron 6272) in these microbenchmarks. In Figure 2.4, large operations (>1024 bytes) over all com-

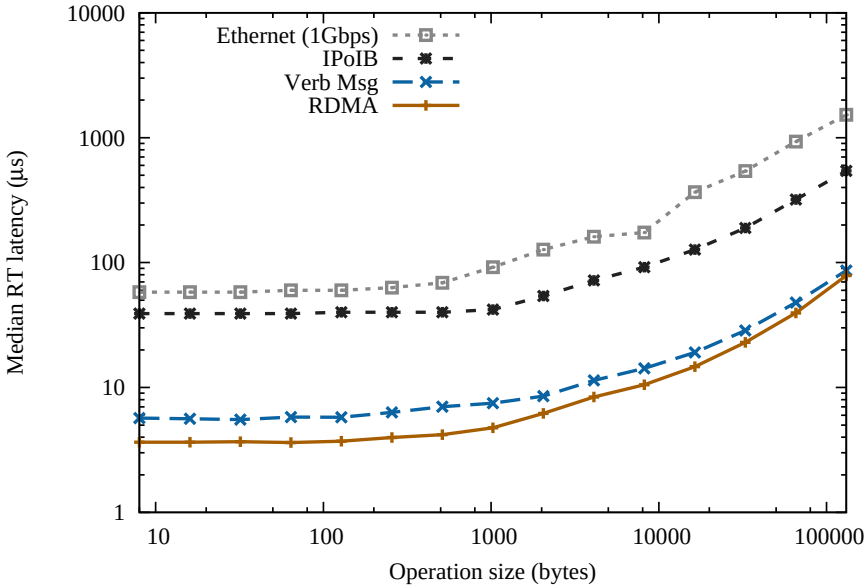


Figure 2.3: Median round-trip latency.

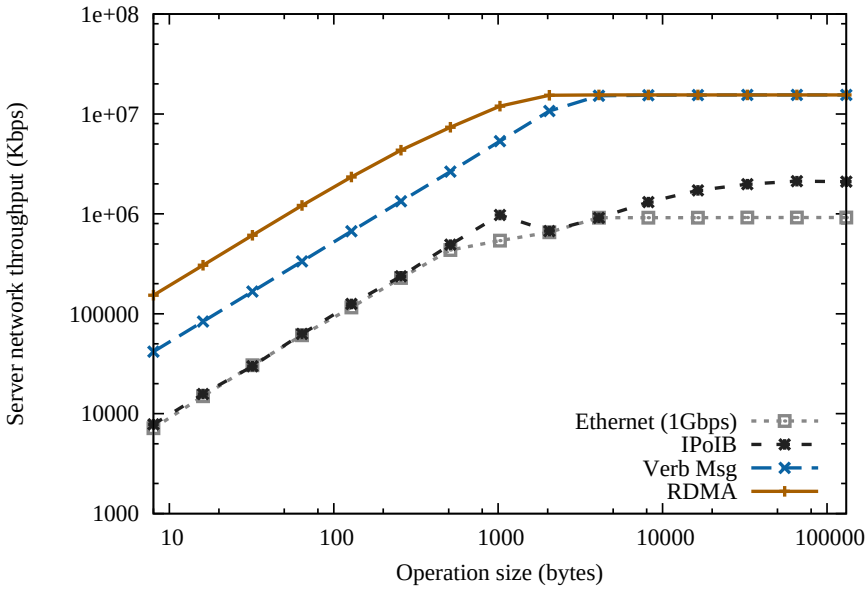


Figure 2.4: Single-server Infiniband network throughput under different communication methods and operation sizes.

munication methods except IPoIB can saturate their respective network's peak throughput. For smaller operations, both RDMA and Verb messages are able to saturate the Infiniband NIC's capacity when running the server on a single CPU core. By contrast, kernel-based transports require more than one core to saturate the network card, hence the much lower throughputs achieved in IPoIB and Ethernet experiments.

RDMA not only incurs zero CPU overhead on the server, it also saturates the network card at the highest throughput. As shown in Table 2.2, a server can sustain 2.45 million operations/second with 16-byte RDMA reads. By contrast, the server can only achieve 0.668 million operations/sec when exchanging RC Verbs request/reply messages. There are two reason for this performance gap. First, each request/reply exchange uses two underlying Verbs messages compared to one for an RDMA read. Second, because there is less bookkeeping for RDMA, our HCA can perform RDMA at a higher throughput (~2.45 million reads) than sending (~700K) or receiving (~1.5 million) Verb messages per second for short messages.

These results are all achieved using the reliable connection (RC) mode, the only transport mode that can issue RDMA read or RDMA atomic operations. In UC mode, acknowledgments are not sent, so message-passing sends are approximately twice as fast. Other evaluations [32] show that unreliable connections and unreliable datagrams achieve more operations per second on the same hardware than reliable connections, but because RC mode is required for RDMA reads and hence client-side searches, this thesis does not explore other transport modes' performance.

RDMA and Verb messaging both scale well when a group of clients simultaneously issues commands to a group of servers.

2.3 Scaleout Performance

We expect to be able to run distributed storage systems in datacenters on dozens, hundreds, or thousands of machines. Therefore, it is important that future datacenter interconnects support high throughput and low latency regardless of the number of servers each client must contact, or the clients connected to each server. The ideal interconnect would offer perfectly scalable throughput and latency, but in the real world, latency increases and per-machine throughput decreases as more connections are added. Overhead of maintaining per-connection state, collisions in the networking fabric, and sub-full bisection bandwidth prevent large networks from reaching theoretical maximum speeds.

To compare Infiniband's scaleout performance to that of an ideal interconnect, we measure the through-

put and latency of 1KB RDMA reads and 128-byte two-way Verb ping-pongs. We use clusters of 1 to 24 servers, utilizing 1 CPU core per machine. Results for our local cluster, used in all Pilaf tests, are presented in Table 2.3b, while Table 2.3a presents results for the Nome cluster, used for most Cell tests. We also tested the PRObE Susitna testbed, used for some Cell tests in Chapter 6, and tabulate the results in Table 2.3c. Because very little per-message processing is performed on the servers, we do not observe a CPU bottleneck. We varied the client count to search the throughput-latency space for RDMA reads, Verb messaging, and simultaneous use of the two (fixed at $\frac{1}{3}$ Verb messages, $\frac{2}{3}$ RDMA reads [1]). Because latency rises with no additional gain of throughput past saturation, and we cover the client count space sparsely, we report the throughput within 5% of the maximum measured throughput with the lowest latency.

On the Nome testbed, we observe constant latency as the number of servers increases for RDMA reads, Verb messages ping-pongs, and a combination of the two. RDMA read throughput and Verb message throughput is almost exactly $24\times$ the throughput on one server, as expected from the constant latency. The hybrid approach achieves a nearly ideal $22.23\times$ speedup from 1 to 24 servers; its peak 24-server throughput of 35.56M ops/sec is 82% of the *aggregate* RDMA and Verb throughput on 24 servers. The results on our local cluster (Table 2.3b) and the Susitna testbed (Table 2.3c) show similar scalability of RDMA and Verb operations. A dearth of machines prevented us from saturating the hybrid approach on our local cluster.

A performance cliff with hundreds of active Infiniband connections per HCA can be observed on Mellanox ConnectX hardware, an effect reported in other work [16]. Figure 2.5 shows the aggregate throughput and mean latency on four server machines, each processing Verb message ping-pongs and RDMA reads from 1 to 188 client processes across 32 additional machines. FaRM's authors ascribe this cliff to the HCA running out of space to cache metadata for each connection. We ran additional experiments and found that if a large number of *inactive* connections were maintained while running experiments on fewer (< 100) active connections, we measured a minimal drop in throughput. We anticipate that future Infiniband networking hardware will repair this limitation by including more on-board memory for caching queue pair metadata. We could have ameliorated this effect by switching to a datagram transport mode for our systems and experiments, since datagram queue pairs are many-to-many (connectionless). We did not pursue this route because we need the RDMA reads that are only available in reliable connection (RC)

[1] This ratio was chosen experimentally, and informed by the maximum RDMA and message-passing ops/sec each HCA can complete.

	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

(a) Nome testbed (ConnectX IB, 20Gbps)

	RDMA read		Verb messaging		Hybrid	
Servers	Throughput	Latency	Throughput	Latency	Throughput	Latency
1	1.63M ops	7.2μs	835K ops	8.4μs	2.32M ops	7.8μs
2	3.18M ops	7.6μs	1.38M ops	9.4μs	4.24M ops†	9.5μs
4	5.97M ops	8.0μs	2.62M ops	11.1μs	5.86M ops†	8.9μs

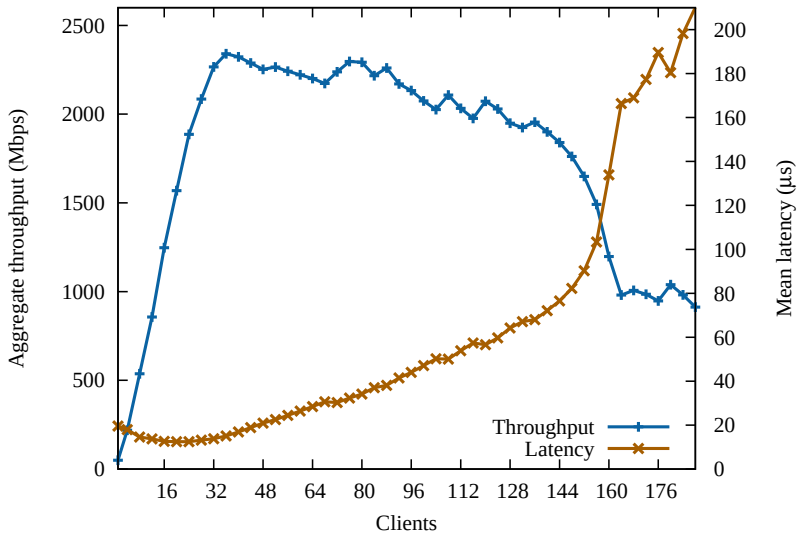
(b) Local testbed (ConnectX VPI, 20Gbps)

	RDMA read		Verb messaging		Hybrid	
Servers	Throughput	Latency	Throughput	Latency	Throughput	Latency
1	1.65M ops	7.3μs	720K ops	6.9μs	2.30M ops	8.4μs
2	3.14M ops	9.2μs	1.27M ops	8.7μs	4.33M ops	8.9μs
4	6.11M ops	9.8μs	2.53M ops	9.9μs	8.16M ops	9.9μs

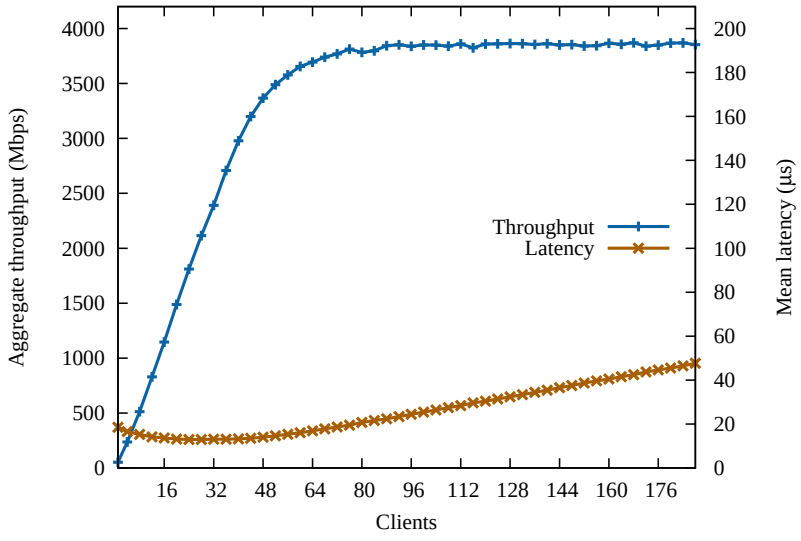
(c) Susitna testbed (ConnectX-3, RoCE, 40Gbps)

Table 2.3: Microbenchmarks of throughput and latency at maximum throughput for 1KB RDMA reads, 128-byte 2-way Verb messages, and the optimal mix of the two. All reported values are within 5% of the peak measured, at minimum latency.

mode.



(a) Verb message ping-pong



(b) RDMA read

Figure 2.5: Throughput and latency of 1KB RDMA reads and 128-byte Verb ping-pongs served by a single core on each of 4 servers with 1 to 188 single-threaded clients connected. Each client connects to all servers. These results are for the Nome testbed, using ConnectX EN DDR (20Gbps) HCAs; above 170 clients, connections failed or were dropped with increasing frequency. Note the two y axes on each graph, and the differing throughput scales between the two graphs.

Infiniband can perform RDMA and message-passing operations at ultra-low latency and high throughput on clusters of servers. As a proxy for any HPC-like datacenter interconnect, it demonstrates the building blocks that will be available to distributed systems in future datacenters.

2.4 Opportunities for System Builders

The low latency of message-passing and RDMA operations combined with the CPU-bypassing capabilities of RDMA suggest high-level opportunities for systems designers. As outlined in Chapter 1, traditional Ethernet-based datacenters impose design limitations on distributed systems, especially the importance of maintaining data-computation locality to minimize round trips. Infiniband and other ultra-low latency, RDMA-capable interconnects remove these limitations.

Computation can be moved to clients to free server CPU resources. With traditional Ethernet-based distributed systems, the performance bottleneck is often the servers' CPUs despite the availability of multiple cores [48]. With ultra-low latency operations made possible by bypassing the kernel and utilizing RDMA, servers can saturate the network using fewer server cores by moving computation to clients. The improvement in CPU efficiency is particularly notable with RDMA, which potentially allows clients to process service requests without involving the server's CPU at all. Efficient CPU usage is crucial in datacenters, which often operate a shared environment by running multiple applications on a single machine [13]. By sharing CPU costs between clients and servers, more applications can be packed onto each machine, fewer machines can be used, and each core can be less powerful [69], yet the same or better performance can be achieved.

Multi-round operations are practical. Because the round trip latency on Ethernet is substantial, traditional systems designs aim to minimize the rounds of communications required to complete an operation. For example, existing key-value stores process each get or put operations in one round trip. With ultra-low latency networking, it becomes feasible to use multi-round protocols without adversely affecting end-to-end operation latency.

It is technically challenging to fully exploit RDMA's performance advantages in a system design. Until recently, the prevalent practice was to use RDMA to optimize verb message exchanges. A more

efficient distributed system design is one in which the locality of data and computation is relaxed, the key **Insight 1** and **Insight 2** presented in Chapter 1. When multiple areas of data can be fetched to the locus of computation with fast network round trips, and computation can be pushed to data with equally fast round trips, many more possible system architectures become available to the designer. This thesis presents the design of two such systems, Pilaf and Cell. Pilaf relaxes all data-computation locality for read-only operations while heeding traditional wisdom about maintaining locality for write operations to prevent excess conflicts and wasted progress. Cell takes a more nuanced approach, selectively relaxing locality when it can yield better performance. The design and implementation of these two systems will reveal the practical challenges that confront a systems architect using HPC-like networks, from the difficulty of guaranteeing consistency in the face of RDMA-CPU memory races to determining when to relax data-computation locality.

We begin by presenting the design of Pilaf, a distributed in-memory key-value store.

3
PILAF DESIGN

Key-value caches and stores are a vital component of today's datacenter-scale applications. Pilaf is a distributed in-memory key-value store built for the datacenters of tomorrow. Following Insight 1, Pilaf relaxes data-computation locality for all read operations. It introduces self-verifying data structures as a solution for detecting memory races in the face of the unsynchronized operations made possible by relaxed locality. Section 3.1 motivate Pilaf's overall architecture. Section 3.2 then explains how clients perform gets using RDMA reads, and Section 3.3 discusses how Pilaf synchronizes clients' RDMA accesses with the server's local memory writes. Finally, Section 3.4 describes the Cuckoo hashing optimization that reduces the number of required roundtrips in the worst case.

3.1 Overview

The most straightforward design would be to take a traditional key-value store and re-implement its messaging layer using Verb messages instead of TCP sockets. However, this design fails to reap the benefits of RDMA, which has much lower latency and CPU overhead than Verb messages. Therefore, our goal is to find a system design that can exploit one-sided RDMA operations without adding too much complexity.

A key-value store has two basic operations: $V \leftarrow get(K)$ and $put(K, V)$, where both the key K and value V are strings of arbitrary length. In our initial design iterations, we tried to use one-sided RDMA operations for both gets and puts. In other words, each client performs RDMA reads to implement gets and RDMA writes to implement puts.

We quickly discovered that using RDMA for all operations leads to complex and fragile designs. First, clients must synchronize their RDMA writes so as not to corrupt the server's memory. Infiniband HCAs support atomic operations (such as compare-and-swap) on top of which one could build an explicit locking mechanism. However, locking over the network not only incurs a performance hit, but also introduces the complication of clients failing while holding a lock. Second, a put operation requires memory allocation to store key-value strings of arbitrary length; such memory management becomes unwieldy in the presence of remote writes. Having clients implement memory management remotely is expensive, with excessive locking and round trips required, and wasted computation expended on conflicts. On the other hand,

letting the server perform some memory management introduces write-write races between the server and clients. Unfortunately, unlike synchronization among concurrent clients, there exists no efficient hardware mechanism to synchronize memory accesses initiated by the CPU and the network card. Last but not least, by making all operations transparent to the server, debugging becomes a painstaking process, as race conditions involving remote accesses are much more difficult to find and reproduce than those involving local accesses.

Our first major design decision is to have the server handle all the write operations (i.e. put and delete) and have the clients implement read-only operations (i.e. get and search) using one-sided RDMA reads. Since real-world workloads are skewed towards reads (e.g., Facebook reported read-to-write ratios ranging from 68%-99% for its active key-value stores [4]), this design captures most of the performance benefits of RDMA while drastically simplifying the problem of synchronization. In fact, the beauty of this design is that it incurs no write-write races, but only read-write races between RDMA reads and the server's local memory writes, Write-write races are the main source of design complexities since they must be avoided at all costs to prevent memory corruption. In contrast, read-write races can be made harmless by *detecting* the presence of such races and re-trying the affected operation. Thus, no fragile and expensive locking protocol is needed.

Figure 3.1 shows Pilaf's overall architecture. Using verb messages, clients send all put requests to the server, which inserts them in its in-memory hash map before sending the corresponding replies. By contrast, gets are transparent to the server in that the clients perform RDMA reads over multiple roundtrips to directly fetch data from the server's memory. As in other key-value store designs [48, 62], the server asynchronously logs updates to its local disk.

3.2 Basic get Operation Using RDMA

We first explain how Pilaf performs gets without involving the server's CPU. We defer the challenge of coping with concurrent puts and gets to Section 3.3.

To allow RDMA reads, the server must expose its data structure for storing the hash table, as shown in Figure 3.2. There are two logical memory regions: an array of fixed-size hash table entries and an extents area for storing the actual keys and values, which are strings of arbitrary length. The server registers both memory regions with the network card, and clients obtain the corresponding registration keys of these

two memory regions (as well as the size of the hash table array) when they first establish a connection to the server. Subsequently, clients can issue RDMA requests to any memory address in these two regions by specifying the memory's registration key and an offset.

In the basic design, a client looks up a key in the hash table array using linear probing [63]. Each probe involves two RDMA reads. The first read fetches the hash table entry corresponding to the key. If the entry is currently filled (indicated by an `in_use` bit), the client initiates a second RDMA read to fetch the actual key and value strings from the extent region according to the address information stored in the corresponding hash table entry. The client checks whether the fetched key string matches the requested key. If so, the get operation finishes. Otherwise, the client continues with the next probe.

3.3 Coping with Read-Write Races

In a traditional key-value store, only the server's CPU can access its memory, so only easy-solvable memory access races between threads are possible. Relaxing data-computation locality introduces read-write races between the CPU and HCA. In Pilaf, local memory writes performed by the server's CPU unavoidably create potential read-write races with concurrent RDMA reads done by clients. This is a challenge as there exists no efficient hardware mechanism to coordinate the CPU and the network card. To inhibit RDMA reads during a write, the server could resort to resetting all existing connections, or temporarily de-register memory regions with the network card. However, both mechanisms are far too expensive to be used for every put operation.

To implement a read operation in any data store, clients need to traverse the server's data structure. The traversal starts from a set of "root" objects with known memory locations and recursively follows pointers read previously. In the context of Pilaf, we can view each hash table entry as a "root" object which points to additional key-value information. Read-write races introduce the possibility that clients can traverse the server's data structure incorrectly.

Two scenarios can result in incorrect traversal. First, a root object can be corrupt. In Pilaf, this happens when the server modifies a hash table entry while a client is reading that entry. Consequently, the client will read a partially modified or corrupt hash table entry, potentially causing it to read the key-value string from an invalid memory location or to incorrectly determine that a key is present or absent. Second, a client's pointer reference can become invalid. For example, in Pilaf, the server may delete or modify an

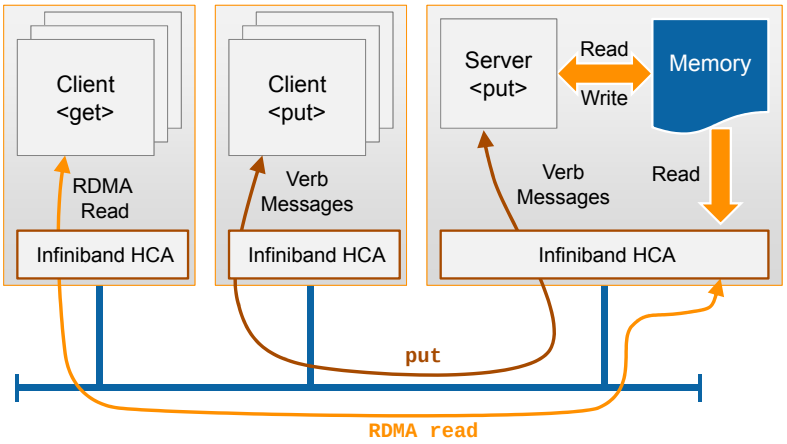


Figure 3.1: Pilaf restricts clients' use of RDMAs to read-only get operations and handles all put operations at the server using Verb messages.

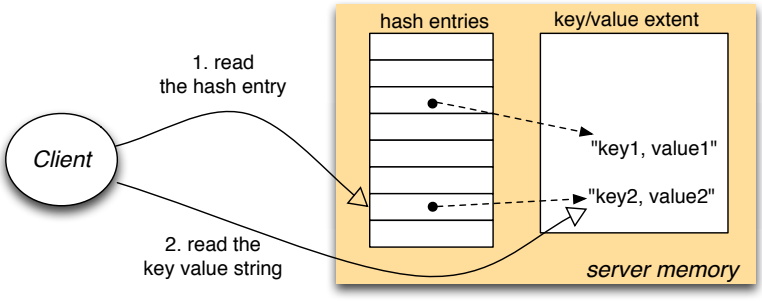


Figure 3.2: The memory layout of the Pilaf server's hash table. Two memory regions are used: one contains an array of fixed-size hash table entries, the other is an extent storing variable sized key-value extents. Clients perform get operations in two RDMA reads, first fetching a hash table entry, then using the address information in that entry to fetch the associated key-value string.

existing key/value pair while a client is holding a pointer reference to the old string from its first RDMA read of the hash table entry. Thus, during its second RDMA access, the client might read garbage or an incorrect key-value string.

To permit correct traversal in the face of read-write races, we introduce the notion of a *self-verifying* data structure by making both root objects and pointers self-verifying. For a root object, we append a checksum that covers the object's entire content. Thus, any ongoing modification on the root object results in a checksum failure. To make a pointer self-verifying, we store it as a tuple combining a memory location, the size of the memory chunk being referenced, and a checksum covering the content of the referenced memory. Therefore, a client can detect the inconsistency between a pointer's intended memory reference and the actual memory contents. For example, if the server de-allocates the memory chunk being referenced and re-uses parts of it later while a client is still holding a pointer to it, the client will fail to verify the checksum when it retrieves the memory content using the pointer. Figure 3.3 shows Pilaf's self-verifying hash table. As a root object, each hash table entry contains a checksum covering the whole entry. The pointer stored in each hash table entry contains a checksum verifying the key-value string being referenced.

Self-verifying data structures ensure correct traversal starting from a set of known root object locations. On rare occasions, the server may need to change the root object locations. This can be accomplished correctly by having the server reset all of its existing RDMA connections to clients to inhibit clients from reading stale root object locations. In Pilaf, whenever the server needs to resize its hash table array, it resets connections so that clients are prevented from performing RDMA reads until the resize is complete. They are allowed to reconnect once the resize operation is complete to obtain up-to-date information about the location and size of the hash table array. Since hash table resizing is infrequent, there is a minimal performance penalty from resetting connections.

A self-verifying data structure allows clients to perform consistent reads in the face of concurrent writes. In addition, the Pilaf server uses a memory barrier to force any updates from the CPU cache to main memory before replying to a put request. Doing so ensures that a subsequent get always reads the effect of any completed puts. As a result, Pilaf provides the strongest consistency semantics, i.e. linearizability [24].

This approach is similar to Merkle Trees [52], which guarantee the integrity of nodes in a hierarchical data structure (like a B-tree). Each parent node includes the hashes (for security) or checksums (for

integrity) of its children's data, which in turn includes the concatenated hashes or checksums of their children's data. Therefore, the integrity of one branch of the tree can be determined by checking hashes or checksums from the leaf to the root, without accessing the rest of the tree. Pilaf's self-verifying structures include a checksum over each root node appended to that node, and a checksum over the destination data for each pointer, permitting the same propagating proof of integrity from a correct (or trusted) root.

3.3.1 Proofs of Atomicity and Serializability

Because the correctness of these technique is subtle, we provide proofs that build from correct RDMA reads to serializable operations. First, checksums are used to guarantee that RDMA reads fetch hash table nodes and hash table extents atomically.

Theorem 3.1. *Let d be some bits of data, with a checksum C_c that can be computed over d. If a stored checksum C_s is concatenated to d, any read of d and C_s unsynchronized with writes to d and C_s can determine that d is consistent iff $C_s = C_c$. Therefore, arbitrary series of writes to d can be made effectively atomic to reads.*

Proof. Consider an ideal b-bit checksum scheme $CS(d) = C_c$. For any data d, changing any bit(s) will yield a different checksum C_c' with probability $p = 1 - 2^{-b}$. The vanishingly small probability $p = 2^{-b}$ of a checksum collision can be ignored for sufficiently large b. Therefore, $CS(d) \neq CS(d')$ for any $d \neq d'$. A self-verifying data element concatenates data d with a stored checksum $C_s = CS(d)$ over that data. Decisive operations (*i.e.*, writes) map d from a good state into some new good state d'', while nondecisive operations (*i.e.*, reads) do not modify d [65]. C_s is only updated to a new stored checksum C_s' once the data d has been modified into some new consistent state d''. Any intermediate inconsistent state can be considered d'. Because $CS(d) \neq CS(d') \neq CS(d'')$, and C_s can only be one of $\{CS(d), CS(d'')\}$, the client can only read some stored checksum C_s corresponding to d or d''. If the data is d', neither possible stored checksum can match the computed checksum $C_c = CS(d')$, and the client will those determine that the data is in an inconsistent state. By definition of atomicity, because the client will only accept d or d'' and will retry reads of intermediate states d', the series of writes that generate intermediate states d' will be effectively atomic. □

Therefore, any data-checksum pair can guarantee that the data read is in an internally-consistent state, whether the pair is a hash table row and associated checksum, or a self-verifying pointer to some variable-

sized data with an associated checksum. With the ability to perform atomic reads and writes on any contiguous data within a data structure, we can consider modifications to a hash table row and associated extents to be a single atomic operation.

Lemma 3.1. *Modifications to hash table rows and linked extents appear to be a single atomic operation.*

Proof. Theorem 3.1 proves that writes to extents segments and to hash table rows are each atomic. In Pilaf, insert operations first create extents, store data in the extents, store the pointer and checksum for the extents in a hash table row, and finally update that hash table row. Therefore, searches will continue to re-read the hash table row from when the first write occurs until the row checksum is updated. At that point, the extents will be atomically available. Similarly, delete operations first corrupt extents, forcing searches to repeatedly re-read the hash table row to try to fetch an updated checksum over the extents, then remove the key from the hash table row before finally updating the row's checksum, at which point the search will terminate and determine the row to be empty. Therefore, a modification to a hash table row and linked extents is a single atomic operation. □

We can now show that operations in Pilaf are serializable. The following definitions are used in this proof:

KeySpace The set of all possible keys that can be stored is the KeySpace.

Inset of a node $inset(n)$ of node n in a general data structure is the intersection of the edgesets for every path from the root to n. In Pilaf, this is the set $(x \mid hash_1(x) = idx(n)) \cup (x \mid hash_2(x) = idx(n)) \cup (x \mid hash_3(x) = n)$.

Outset of a node $outset(n)$ of node n is the union of all of the edgesets leaving n. The outset of Pilaf nodes is $\varnothing$.

Keyset of a node The keyset is the edgeset of everything in $inset(n)$ but not in $outset(n)$; that is, $keyset(n) = inset(n) - outset(n)$. In Pilaf, nodes have no outset, so $keyset(n) = inset(n)$.

To show serializability, we follow the technique outlined in [65]. First, we show that each operation in Pilaf maps a good state to a good state, following GS1 through GS3 from [65].

Proposition 3.1. *All decisive (write) operations in Pilaf map a good state to a good state. For all search operations, the correct key-value pair for some serial ordering can be found if the inset of the the associated hash table row has not changed between the client reading the hash table row and associated extents.*

Proof. As outlined in [65], three invariants must be maintained to guarantee every state is a good state. Each invariant is embodied in one of the three claims below. By Lemma 3.1, extents need not be considered separately in this proposition.

Claim 1. If key x is in node n, then x is in the keyset of n.

At each point in the life of a Pilaf distributed hash table, each row n either holds a valid key for that row (*i.e.*, $hash_i(x) = idx(n)$ for $i \in [1, 3]$) or no key. Insert operations only put x in one of the 3 possible rows for that key, and insert operations that involve a kick-out can also only move keys from one possible row to another possible row for that key. Delete operations therefore can also only remove x from one of its 3 possible rows.

Claim 2. The union of all hash table rows in Pilaf is the KeySpace (*i.e.*, all possible keys). All hash table rows do *not* partition the KeySpace; instead, they cover the KeySpace exactly $N = 3$ times for 3-way Cuckoo hashing.

Unlike the structures in the serializability techniques outlined in [65], the hash table rows in Pilaf do not partition the KeySpace. This does not compromise the effect of the original invariant: to guarantee that the location for a key x is unambiguous (in the invariant in [65], exactly one possible node $n \mid x \in keyset(n) \forall x$). The location of a key x in Pilaf is similarly unambiguous: Every possible key x hashes to exactly $N = 3$ hash table rows, so the inset of each hash table row is the set of all keys that hash to that row's index.

Claim 3. If the search for key x is at node n, then $x \in keyset(n)$ as well as the keysets of the other $N - 1$ nodes n where $hash_i(x)$ for $i \in [1, 3]$. In addition, the reads of the three possible nodes where $x \in keyset(n)$ can be made effectively atomic.

Searches use well-defined hash functions that deterministically map x to exactly $N = 3$ possible hash table rows. To guarantee that x be found if it exists in the hash table when a search occurs in some serial

ordering, the reads of the three hash table rows must be made effectively atomic. Section 3.4 outlines one possible solution using additional client-side reads plus an additional version number field in each hash table row.

Conclusion. Any search through a Pilaf store for x, if it terminates, will terminate in the correct place: in a node whose keyset contains x.

By providing effective atomic reads of all N possible hash table rows that could contain x, the search will terminate correctly either in the node n containing x, or having determined that none of N nodes contain x. Therefore, searches always terminate correctly. □

Therefore, every modification of the Pilaf hash table maps a good state to a new good state. Because this is the case, all Pilaf operations can be shown to be serializable.

Theorem 3.2. *Every Pilaf operation maps a good state to a good state. Therefore, by the give-up theorem [65], it is serializable.*

Proof. As shown in Proposition 3.1, all Pilaf operations map a good state to a good state. In addition, all dictionary actions in Pilaf computations follow the give-up technique guidelines. Therefore, by the give-up theorem, Pilaf operations are serializable [65]. □

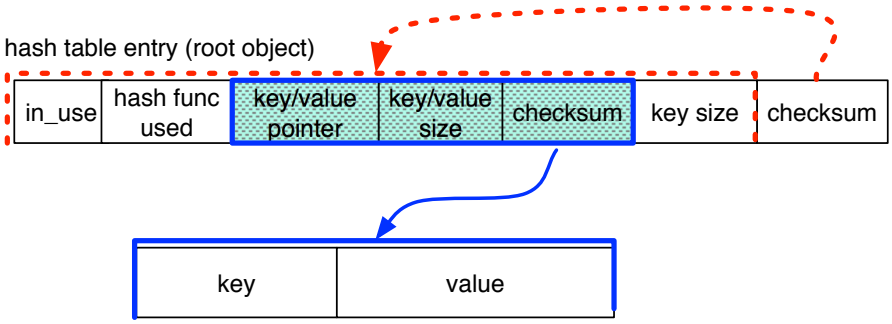


Figure 3.3: *Self-verifying* hash table structure. Each hash table entry is protected by a checksum. Each entry stores a self-verifying pointer (shown in shaded fields) which contains a checksum covering the memory area being referenced.

3.4 Improving a Hash Table's Memory Efficiency

In the basic distributed hash table design, a client performs linear probing to look up a key in the server's hash table array. This simple hash scheme does not achieve a good tradeoff between memory efficiency and operation latency. For example, when the hash table is 60% full, the maximum number of probes required can be as high as 70. To achieve good memory efficiency with fewer probes, Pilaf uses N-way Cuckoo hashing [58, 36]. This hashing scheme uses N orthogonal hash functions, and every key is either at one of N possible locations or absent. If all N possible locations for a new key are filled, the key is inserted anyway, kicking the resident key-value pair to one of that key's alternate locations. That operation may in turn kick out another pair, ad infinitum. The table is resized when a limit is reached on the number of kicks performed or when a cycle is detected.

The main challenge in using Cuckoo hashing for Pilaf lies in the process of moving an existing entry to a different hash table location. Ordinarily, bulk key movements such as resizing the hash table require that the server reset all existing RDMA connections. This is not desirable, as the need to move a key occurs much more frequently than table resizing with Cuckoo hashing. Without resetting connections, there is the danger that a key-value pair might appear to be "lost" to the clients while the server is moving it to a new location. To address this issue, during a put operation the server first calculates the new locations of every affected key without actually moving the keys. Then, starting from the last affected key, the server shifts each key to its new location, thereby ensuring that a key is always stored in at one or two (instead of zero or one) hash table entries during movement. Algorithm 2 provides pseudocode for this process, showing how the server reads forward along the chain to find all keys to move, then backs up the chain to move the keys.

However, even with this modification, possible interleavings exist that rarely compromise serializability. A series of at least two insert requests may cause kick-outs that overlap the hash table rows the search is probing. When key K is kicked from the sth possible location for that key to a new possible location $s' < s$, then a concurrent search may incorrectly determine that K is not present in the hash table. If the search checks locations $[1, s-1]$, then the kick-out occurs, then the search completes its check of locations $[s, N]$, K will not appear. If an application requires serializability, we modify the search operation to probe at most $2N-1$ locations (rather than N locations) before declaring a key absent. We must also add a short version number (*e.g.*, 8 or 16 bits) to each hash table row, incremented each time that the hash

Algorithm 1 Finding a key in Pilaf's hash table, including an optional extension that probabilistically guarantees serializability.

```
1:  function SEARCH(k, serializable)          ▷ Find value v for key k, if present
2:      version ← [N]
3:      for i ← 1 to N do
4:          if table[hash(k, i)] is !empty then
5:              p ← table[hash(k, i)].pointer
6:              k', v ← extents[p]
7:              if k == k' then               ▷ Multiple keys can hash to the same row
8:                  return v
9:          versions[i] ← table[hash(k, i)].version   ▷ Omit when not guaranteeing serializability
10:     if !serializable then                 ▷ Save RDMA reads by skipping rare case
11:         return nil
12:     for i ← 1 to N − 1 do
13:         if versions[i] != table[hash(k, i)].version then
14:             go to 2
        return nil
```

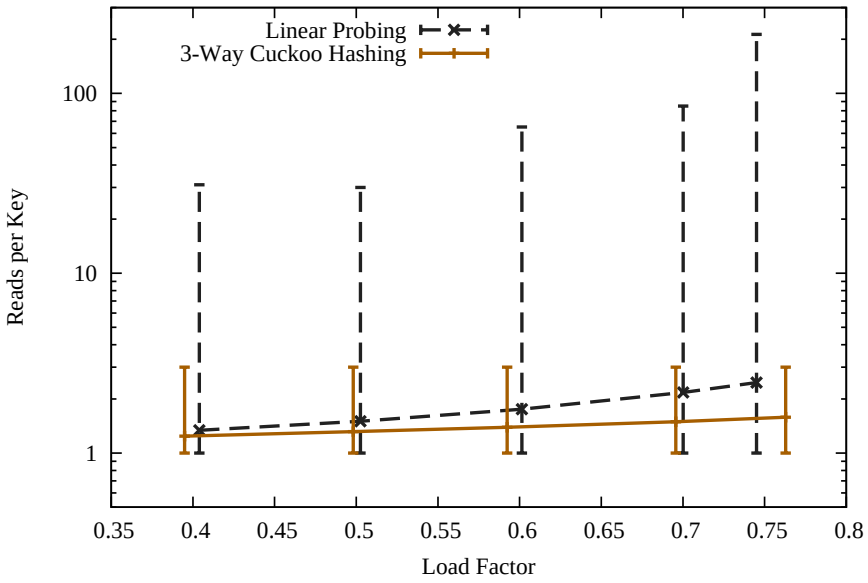


Figure 3.4: The average number of probes required during a key lookup in 3-way Cuckoo hashing and linear probing. The error bars depict the median (rather than minimum) and maximum values.

table row is modified. Algorithm 1 shows how a Pilaf Cuckoo hash table search works with this change.

The second set of reads is necessary to determine if the hash table rows probed earlier remain consistent while the later rows are probed. If during the second pass, any row contains a different version number

Algorithm 2 Inserting into Pilaf's N-way Cuckoo hash table, including the RDMA-aware chained kick-out process that maintains the accessibility of every key being moved.

```
1:  function INSERT(k, v)
2:      for i ← 1 to N do
3:          if table[hash(k, i)] is empty then
4:              table[hash(k, i)] ← k, v, i
5:              return
6:          else if table[hash(k, i)].key == k then
7:              table[hash(k, i)].value ← v
8:              return
9:      traceback, hashback ← []                          ▷ All slots for k are in use
10:     foundslot ← -1
11:     tempkey ← table[hash(k, 0)]
12:     hashidxend ← table[hash(k, 0)].hash
13:     repeat
14:         hashidxstart ← (hashidxend + 1)%N
15:         for i ← hashstart to hashend − 1 do
16:             if table[hash(tempkey, i)] is empty then
17:                 foundslot ← i
18:         traceback.append(hash(tempkey, hashidxend))
19:         if foundslot then
20:             hashback.append(foundslot)
21:         else
22:             hashback.append(hashidxstart)
23:         if foundslot then break
24:         hashidxend ← table[hash(tempkey, hashidxstart)].hash
25:         tempkey ← table[hash(tempkey, hashidxstart)].key
26:     until foundslot or cycle found or max kick-outs reached
27:     if foundslot then
28:         target ← hash(tempkey, foundslot)
29:         for source in traceback, hashidx in hashback do
30:             table[target] ← table[source]
31:             table[target].hash ← hashidx
32:             target ← source
33:         table[hash(k, 0)] ← k, v, 0
34:     else
35:         table.Resize()
36:         table.Insert(k, v)
```

than was read by the first probe, then some subset of hash rows for K has been changed, the search may have missed the presence of K, and the search must be repeated. $2N$ probes are not required because migration within the set of possible hash table rows is commutative; *i.e.*, if a key was removed from the Nth location as part of a kick-out, it must have been placed in one of the remaining $N - 1$ locations. Like the guarantees on checksum-based self-verifying data structures, this serializability is probabilistic. Given b-bit version numbers, if 2^b modifications are made to a particular hash table row between the first and second set of reads, the version number will appear unchanged. In practice, there is a vanishingly small probability that 2^b writes that cause a kick-out to the same hash table rows for a reasonably large b. For example, $b = 42$ is sufficient to store a Unix epoch timestamp with microsecond granularity, which would not roll over for 136 years. A timeout-based guarantee can remove the probabilistic quality of this argument: clients can consider the second pass correct only if for a per-kickout CPU time t, less than $t \cdot 2^b$ elapses between the end of the first and second sets of probes.

Different parameter values for N were explored, and it was determined that 3-way Cuckoo hashing achieves the best memory efficiency with few hash entry traversals per read. As Figure 3.4 shows, at a fill ratio of 75%, the average and maximum number of probes in 3-way Cuckoo hashing is 1.6 and 3, compared to 2.5 and 213 respectively for linear probing. We explored 2-way and 4-way Cuckoo hashing variants: the former had poor memory utilization, while the latter offered minimal fill ratio improvement for a modest increase in average reads per key. 3-way Cuckoo hashing yielded a maximum fill ratio of 78%, with 1.35 average reads per key at the average utilization of 60%. Although achieving these higher fill ratios required a fraction more average RDMA reads to locate each key's row in the hash table, plus an additional RDMA read to fetch the extents, the low latency and total CPU cost associated with each read made 3-ary Cuckoo hashing an efficient choice for both time and memory.

3.5 Summary

Designing distributed storage systems like key-value stores for the datacenter interconnects of tomorrow will require awareness of new opportunities and obstacles. Pilaf demonstrates design choices for a high-performance key-value store over Infiniband, especially heeding Insight 1, that distributed systems with relaxed data-computation locality are now practical. By using client-issued RDMA operations for all reads, performing all writes at the server using message passing, and utilizing self-verifying data struc-

tures, Pilaf achieves high performance with practical complexity. The next system, Cell, refines Insight 1 into Insight 2: *selectively* relaxing data-computation locality yields optimal performance and load balancing.

4
CELL DESIGN

Like key-value stores, sorted stores are an important component of datacenter-scale applications. Today's popular sorted stores are designed around the limitations of Ethernet interconnects, but datacenter networks with HPC-like features will open up a new design space. This thesis presents a distributed in-memory sorted store around Insight 2: selectively relaxed data-computation locality yields better performance and load balancing. Cell is a distributed in-memory B-tree with client-side and server-side reads. This chapter gives an overview of Cell in Section 4.1, then discuss the main components of our design: our B-tree structure in Section 4.2 and our hybrid search technique in Section 4.3 and Section 4.4.

4.1 Overview

Traditional distributed B-tree stores spread an opaque tree across many machines [8]. Clients issue requests to servers to traverse a portion of the tree, and servers return a pointer to where the traversal should continue. To minimize Ethernet round trips, as much related B-tree structure as possible is located on each server. Cell organizes data in a hierarchical structure: a B-tree of B-trees. As we will see, this approach ensures that both server-side and RDMA-based searches are efficient. At the cluster level, Cell builds a B-tree out of fat nodes that we call *meganodes*, containing tens or hundreds of megabytes of structural metadata. Meganodes are stored in-memory and spread uniformly across servers in the cluster. Within each meganode, Cell builds a local B-tree consisting of small nodes (e.g. 1 kilobyte). The local B-tree serves two purposes: (1) it allows a server to search efficiently for a key within a meganode, and (2) it allows remote clients to search within a meganode using a small number of efficient RDMAs. Other systems, e.g. BigTable [8], do not have the second requirement. As such, they can use different searchable local data structures such as skiplists that 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 meganodes store remote pointers to other meganodes on remote machines. All pointers consist of a region ID and offset. A region is a large contiguous memory area (e.g. 1GB), allocated independently by each server, that can hold either meganodes or key-value data (we call the latter an *extents region*). In order to find each region, all clients and servers maintain

a cache of the global region table that maps region IDs to the IP addresses and ports of the responsible server processes. Servers also actively exchange region information with each other asynchronously. We assume that server membership is maintained reliably using a service like Zookeeper [26].

As shown in Figure 4.1, 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 [16] or Pilaf [54]. However, we minimize this complexity by carefully designing our B-tree, discussed next.

4.2 B-tree Operations

Cell uses a type of external B-tree called a *B-link tree* [38]. We use the same structure at both the meganode scope and within each meganode, as illustrated in Figure 4.2. B-link trees offer much higher concurrency than standard B-trees due to two structural differences: first, each level of the tree is connected by right-link pointers, and second, each node stores a *max key* which serves as an upper bound on the keys in its subtree. (We additionally store a min key to cope with concurrent RDMA reads; see Section 4.3.) Sagiv [61] refined the work of Lehman and Yao [38] to yield 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 [28, 47].

We follow Sagiv's algorithm when operating within a meganode; we summarize his algorithm here briefly. 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. Until that insert completes, the new node can still be reached by the right link from its left sibling.

Deletions simply remove the key from the appropriate leaf node under a lock. Sagiv also suggested

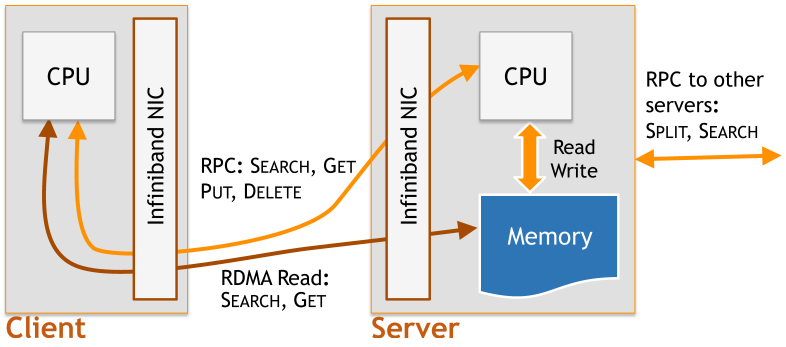


Figure 4.1: Cell's architecture and interactions. See discussion for details.

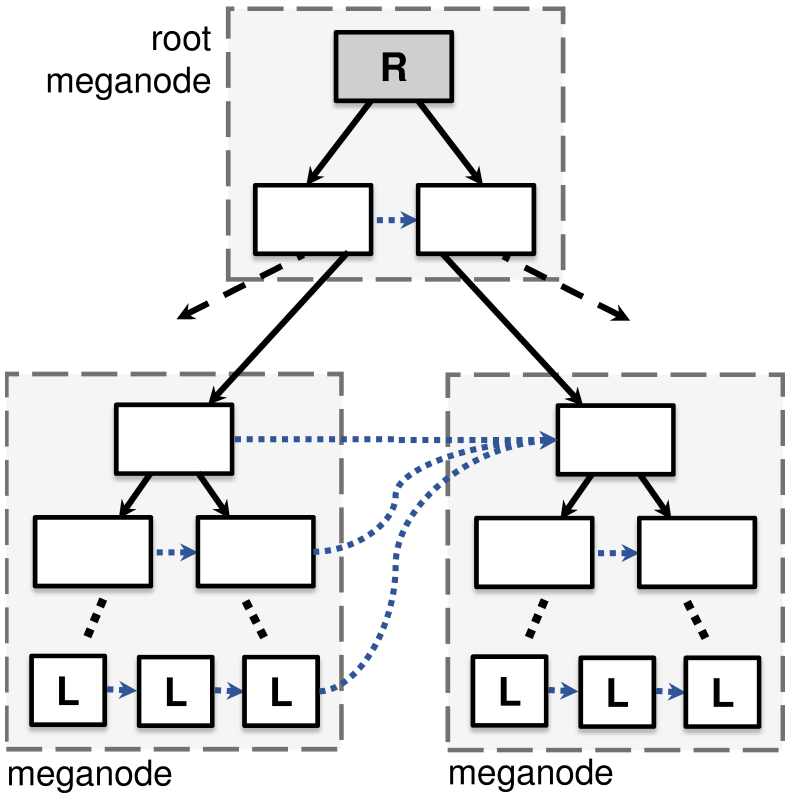


Figure 4.2: The structure of Cell's data store, a B-link tree of B-link trees. Each individual meganode contains a complete B-Link tree with a root (R) and level-linked nodes at each level. The meganodes are also level-linked. The leaves (L) of the bottommost meganodes point to key-value data stored in the local extent region of that machine.

a compression scheme to deal with underfilled nodes left by deletions, but this requires locking three nodes at a time. Although we could easily implement his scheme within a meganode because all locks are local, we choose to avoid deletion rebalancing altogether. In fact, many existing database systems, such as Berkeley DB [56], avoid deletion rebalancing to improve concurrency [20]. Sen and Tarjan [64] also showed that this practice has provably good worst-case properties.

We now describe how to extend Sagiv's algorithm to our "nested" meganode structure. We discuss the server-side operations: search (and get), put, and delete. Client-side search, which is completely decoupled from server-side processing, is discussed in Section 4.3.

(Server-side) Search. To search for a key-value entry, clients iteratively traverse the tree one meganode at a time by sending search requests to the appropriate server or servers. The first request is sent to the server containing the root node R (see Figure 4.2). This server uses Sagiv's algorithm as described above to search within the root meganode, until it reaches a pointer (in a leaf node) that is not local to the machine. This remote pointer is returned to the client, which sends the search request to the server owning the target node, and the process repeats. When a leaf meganode is reached, the server performing the search finds a pointer to the key-value data in its local extent region, and returns this data to the client. To bootstrap the search operation, we ensure that a pointer to R is always stored at offset 0 in the region with lowest ID across the cluster.

We can speed up the search by caching the results returned by meganode searches close to the root. This type of caching is effective because updates in a B-tree such as ours occur exponentially infrequently with a node's height in the tree [64]. Since the tree is very shallow at the meganode level (usually at most height 3), clients only cache results from the root meganode. Specifically, the server traversing the root meganode returns, in addition to a remote pointer, the key range covered by that pointer. This range is determined by looking at the key to the left and right of the pointer (which could be the min or max key). If a cached remote pointer leads the client to the wrong node—which it can detect by looking at the node's max key—it simply follows right-link pointers until we arrive at the correct node. Thus, the right-link strategy used to handle concurrency in B-link trees also gives us a strategy for caching. It even tells clients when to invalidate a cached entry: if a search ever traverses a right-link pointer, then any information cached from the parent must be stale, since otherwise the information in the parent node would have sent the client to the correct node directly.

MEGANODE-SPLIT(X):

1. Identify a split key in the root node of X, which yields subtrees X_{left} and X_{right}. Wait for X_{right} to stabilize:
   1.a. Prevent new updates to X_{right} and wait for existing updates to complete.
   1.b. Lock the range of X_{right} for updates. Updates to X_{left} are still allowed.
2. Asynchronously copy X_{right} to a new meganode:
   2.a. Select a server at random and request a free meganode. Repeat until a server responds with a free, unused meganode.
   2.b. Copy nodes to the new meganode asynchronously:
      2.b.1. Lock the root node and split off the portion containing X_{right} (then release the lock).
      2.b.2. Copy the remaining nodes, replacing the local region ID in all local pointers with the region ID of the target meganode. (Remote pointers should not be changed.)
3. Update the right-link pointers of X_{left}. For each level in the tree:
   3.1. Traverse right-link pointers until reaching a node whose min key exceeds the split key.
   3.2. Lock the node. If it is no longer the rightmost node, unlock and repeat step 3.1. Otherwise, update the right link to point to the root of the new meganode containing X_{right}.
4. Mark the local nodes of X_{right} as invalid, and release the range lock.

Figure 4.3: Protocol for splitting a meganode across servers.

Deletion. To delete a key-value entry, first perform a server-side search for the key. Then, perform a deletion on the leaf node according to Sagiv's algorithm above. As keys are deleted and individual nodes become empty, the meganode itself may become underfilled or empty. However, we avoid deletion rebalancing at the meganode level similarly to the node level. The reason is more fundamental this time: deletion rebalancing requires locking entire meganodes potentially spread across multiple machines, requiring distributed locks and significantly reducing concurrency.

Insertion. To insert (or update) a key-value entry, first perform a server-side search for the key. Then, perform an insertion on the leaf node according to Sagiv's algorithm above. As keys are inserted and individual nodes split, the meganode itself may become full. Unfortunately, unlike an empty (mega)node, a full (mega)node cannot simply be ignored.

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-grained protocol inspired by Sagiv's algorithm that allows greater concurrency, shown in Figure 4.3. The protocol 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 meganode (possibly on a remote server) asynchronously. Then, it updates 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 4.2. 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 discussed in the next section.

4.2.1 Proof of Serializability

Before proceeding, it is important to establish that the three dictionary operation to be applied to Cell (search, insert, and delete) are serializable. To do so, the the procedure from Theorem 3.2 in Section 3.3.1 is repeated. We first show that each operation maps Cell from a good state to a good state. Then, because Cell satisfies the requirements for both the link and give-up theorems in [65], its operations are serializable. A number of definitions are necessary to clarify this proof; the definitions in Section 3.3.1 are augmented by the following:

Edgeset of a pointer The edgeset of a pointer p is defined by separating keys s_1 and s_2 such that $s_1 < xs_2 \forall x \mid x \in edgeset(p)$. The edgeset of a pointer in Cell is a superset of $keyset(n)$ of the pointer's target node n.

Edgeset of a rightlink Each Cell node n has a rightlink with edgeset $x \mid x > maxkey(n)$.

Inset of a node Because the leftmost node at each level of Cell's B-link tree can be considered a root, because the inset of each such node is the KeySpace. Therefore, $inset(n)$ in Cell is the union of the intersection of the edgesets from each root (*i.e.*, the leftmost node at the same and higher levels as n) to n.

Outset of a node $outset(n)$ of Cell node n is the union of all of the edgesets leaving n, both rightlinks and edges to children.

One lemma is necessary to establish the proof of serializability for Cell's operations.

Lemma 4.1. *Searches for x during concurrent Cell node splits arrive at node n with $x \in inset(n)$ (defined by the separating keys s_1 and s_2 on the pointer to n) such that either $x \in keyset(n)$ or $x \in outset(n)$ via at least one rightlink traversal.*

Proof. In the absence of node splits, searches for x arrive at n where $x \in inset(n)$ and $x \in keyset(n)$. That is, the separating keys s_1 and s_2 on the inbound edges match $minkey(n)$ and $maxkey(n)$ stored within n that define $keyset(n)$. Node splits appear as two atomic operations to searches:

1. Node n is replaced by a pair of nodes n and n', connected by a rightlink, that cover the keyset of the original node. Both nodes are only accessible via the link from the parent of n to n (and subsequently the rightlink from n to n' if necessary).

2. A link from the correct parent of n' to n' is added, so that subsequent searches for keys x in $keyset(n')$ that traverse to $parent(n')$ will proceed directly to n'.

Additional node splits may separate n and n' by additional nodes before any parent-child links are created, but n' remains accessible from n (*i.e.*, $x \in outset(n)$) via the chain of rightlinks. □

Proposition 4.1. *All server-side operations in Pilaf map a good state to a good state. All server-side and client-side search operations can either continue a traversal correctly at any point or backtrack to a correct intermediate node, and all searches terminating at the leaf level terminate in the correct location. Therefore, all states visible to local or remote searches are good states.*

Proof. As outlined in [65], three invariants must be maintained to guarantee that each operation maps a good state to a good state, a necessary prerequisite to show serializability. Each invariant is embodied in one of these three claims:

Claim 1. If key x is in node n, then x is in the keyset of n.

The B-link tree that forms Cell's sorted structure holds all keys at the leaf level. Each leaf node maintains a min key and max key, defining the range (keyset) for that node. Insert and delete operations check the keyset of a leaf node n before performing the insertion or deletion of key x, and continue the search for the correct node if $x \notin keyset(n)$ (*i.e.*, $minkey \leq x$ or $maxkey >$x). Server-side searches similarly check the range of a leaf node before determining if x is present or absent in that node. Indeed, this procedure is performed at *each* level of the tree, and if $x \notin outset(n)$ via its child links for some

node n at any level of the tree, the search will proceed to a rightlink (if $x \in outset(n, rightlink)$) or backtrack to a higher level, up to the root.

Claim 2. The keysets of the leaf nodes in Cell partition the KeySpace.

The leaf level of the Cell store begins with a min key equal to the smallest possible key, and a max key equal to the largest possible key; that node covers the entire KeySpace. Insert operations may split the node, each time node n splits into n' and n'', $minkey(n) = minkey(n')$, $maxkey(n) = maxkey(n'')$, and $maxkey(n') = minkey(n'')$. Delete operations delete individual keys, but do not delete nodes, and do not change the min key or max key for any node. Search operations do not modify global state. Therefore, the keysets of the rightlinked-list of leaf nodes always perfectly partitions the KeySpace.

Claim 3. If the search for key x is at node n, then x is in $keyset(n)$ and n is a leaf node, or there is a path from n to a node m where $x \in keyset(m)$ and every path from n to m has x in its outset.

By Lemma 4.1, a search for x concurrent with write operations arrives at $n \mid x \in inset(n)$, then either $x \in keyset(n)$ or $x \in outset(n)$ via its rightlink. If the search arrives at n but $x \leq minkey(n)$ or $x > maxkey(n)$, then the search will go to an ancestor n' of n (a node the search for x has already visited) and retry from n'. In the face of *meganode* splits, when nodes can be invalidated and reused, it is possible for searches to reach some n such that $max(keyset(n)) < x$. The proof of serializability in the face of meganode splits is presented in the next section.

Conclusion. Any search through a Cell store for x, if it terminates, will terminate in the correct place: the node $n \mid x \in keyset(n)$.

Claims 1 and 2 prove that the leaf nodes in a Cell tree partition the KeySpace, and that any key x can only be found in the node with a keyset (defined by min and max keys) containing x. Claim 3 guarantees that a search for x currently at node n either finds $x \in keyset(n)$ or can reach the node n' with $x \in keyset(n')$. Therefore, every modification of the Cell B-link tree maps a good state to a new good state. □

Because every operation maps the Cell tree from a good state to a good state, all Pilaf operations can be shown to be serializable.

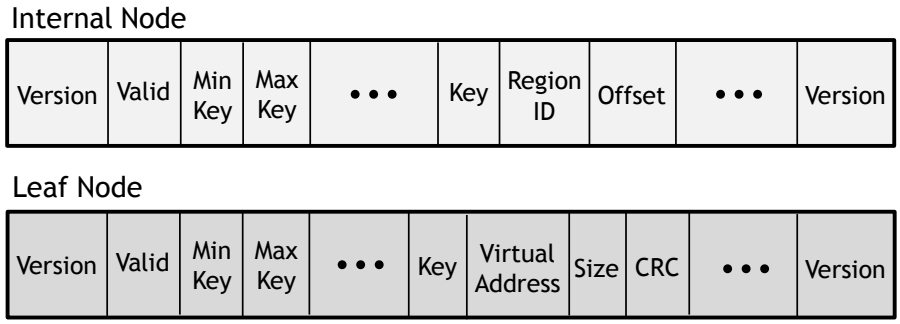


Figure 4.4: The structure of internal and leaf meganodes in Cell's B-link trees. Each node packs two matching versions, a minimum and maximum key, and zero or more pointers to other nodes or to extents memory in a block of a few kilobytes.

Theorem 4.1. *Every Pilaf operation maps a good state to a good state. Therefore, by the give-up theorem [65], it is serializable.*

Proof. As shown in Proposition 3.1, all Pilaf operations map a good state to a good state. In addition, all dictionary actions in Pilaf computations follow the give-up technique guidelines. Therefore, by the give-up theorem, Pilaf operations are serializable [65]. □

4.3 Client-side Search Using RDMA

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 stored at the server. At the end of the search, the client performs an additional RDMA read to fetch the actual key-value data from the server's extent region. Alternatively, if this search is part of an insertion or deletion, the client issues an insert or delete request to the server containing the leaf node returned by the search. Cell servers check the correctness of the suggested leaf node using its min/max key and perform the actual writes to the tree nodes and key-value data.

A full-sized 64MB meganode built from 1KB small nodes contains a 5-level local B-link tree when keys averaging 36 bytes are used. 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 (~15 microseconds), so the overall client-side search latency remains small despite

the extra round trips. (2) The latency overhead pales in comparison to the queuing delay if the server CPU is bottlenecked at high load. To reduce search round trips and ameliorate hotspots at the root meganode, clients cache fetched nodes. We follow the same strategy as for server side search: only cache nodes that are high enough in the tree, and if we ever follow a right-link pointer, invalidate any information cached from the parent node.

Permitting client-side RDMA reads introduces subtle concurrency challenges with respect to server-side tree modifications. The first concurrency challenge lies in the reading and writing of individual nodes. Sagiv's algorithm requires reads and writes of each node to be atomic. This is easy to enforce among server-side operations using CPU synchronization primitives. However, no such primitives exist to synchronize between RDMA reads and the server's memory access on the hardware used for most datacenter servers. To ensure that RDMA reads are consistent, we use two techniques (see Figure 4.4):

- Small B-tree nodes are fixed-size and have clearly delineated boundaries. We bookend nodes with version numbers, and use memory barriers to ensure that the two version numbers are equal and even (*i.e.*, the least-significant bit is reset) in main memory only when the node contents are valid.
- Key-value entries, in contrast, have variable size, and have no fixed boundaries. A CRC over the key-value extents for each pair is stored in the leaf level of the B-tree.

The correctness of each of these schemes in the face of all possible RDMA read/server write interleavings is subtle.

4.3.1 Concurrency in B-Tree Nodes

To protect small B-tree nodes, Cell uses a version number stored at the head and tail of each small (1KB) node. These version numbers are incremented before and after node modifications, so that even version numbers (*i.e.*, values with the least-significant bit reset) correspond to unlocked nodes, and odd version numbers indicate locked nodes. If RDMA reads and CPU writes could be ordered, RDMA reads might see the node as looking like one of the examples in Figure 4.5. However, no such ordering is possible, so the server proceeds through three steps to modify a node:

1. Increment both version numbers to the same odd number. Flush the CPU cache to main memory via a memory barrier to ensure that the version number changes are visible before any node contents are

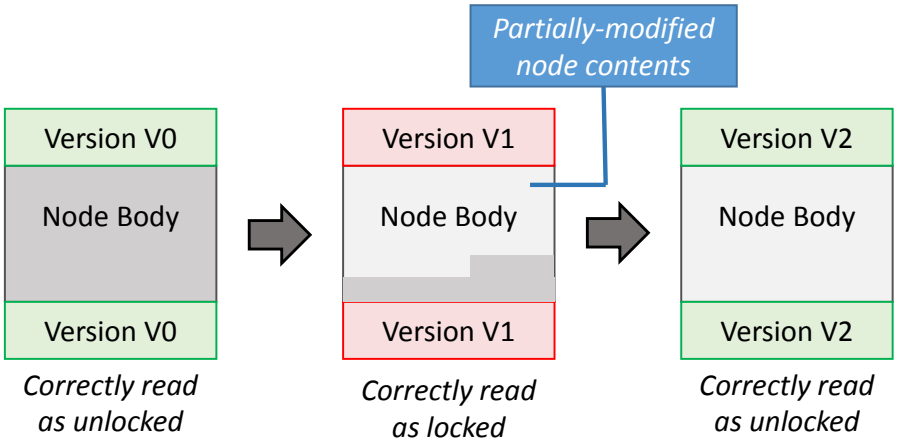


Figure 4.5: Three possible ways a node can appear when an RDMA read fetches the node around a write. It can be fetched unlocked, before any writes occur (Version V0); locked, while the node contents may be in flux (Version V1); unlocked, after all writes complete and the CPU cache is flushed to main memory (Version V2).

modified.

2. Perform the necessary updates to the node's contents and invoke another memory barrier. Any changes will therefore be visible in main memory before the node is unlocked.

3. Increment both version numbers to the same even number. A memory barrier is *not* necessary after this step, but we perform one to reduce the delay before node changes can be read by clients.

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 number, indicating that the RDMA should be retried. Figure 4.6 demonstrates some of the possible views that interleaved reads and writes can yield. This method works because RDMAs are performed by the NIC in increasing address order[1] and because individual nodes have fixed internal boundaries. This technique guarantees that clients will read B-tree nodes atomically. Because correct, atomic client-side reads are indistinguishable from atomic server-side reads, Theorem 4.1 suffices to prove that serializability is maintained.

Concurrency challenges also occur during a meganode split. Although the server can block its own operations during the split, it cannot block clients' RDMA reads, so we must ensure the latter remain

[1] Our scheme works for NICs that read in strictly increasing or decreasing address order. To handle NICs that read in random order, we need to adopt the solution of FaRM [16] of using a version number per cache line, or fall back to CRCs over each node's contents.

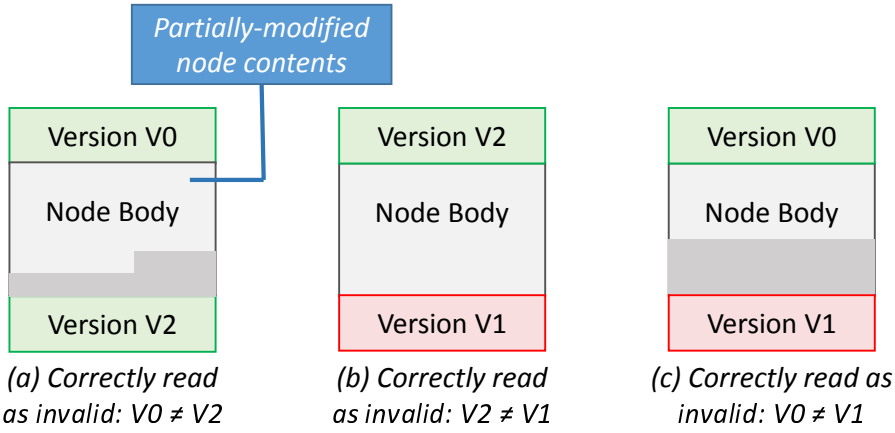


Figure 4.6: RDMA reads remain correct in the face of concurrent server-side writes. Read/write interleaving could cause the RDMA read to see any of these apparent node states, each of which is correctly detected as invalid.

correct. The problem occurs during node invalidation and reuse (step 4 in Figure 4.3). By setting an explicit invalid flag 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 [2]. This is why it is crucial to store the min and max key range in a node: it allows the client to detect that the search has gone astray (*i.e.*, the probe key is not within the $(minkey, maxkey]$ range) and restart it.

4.3.2 Proof of Serializability

The serializability of operations during meganode splits is subtle, and requires an additional proof. One additional lemma provides necessary structure for this proof.

Lemma 4.2. *The min key, max key, and* `invalid` *bit 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.*

Proof. Claim 2 guarantees that a single possible node n for key x exist at any time, to make the presence or absence of x in the global KeySpace unambiguous. Intuitively, to maintain this unambiguity, any ac-

[2] Because nodes have fixed boundaries, client reads cannot fall in the middle of a node

cessible copy n' of n must maintain the same $keyset(n') = keyset(n)$ and contain identical contents. Cell's meganode split algorithm creates one copy n' of each original node n to be moved from M_{right} of meganode M to a new meganode M', as outlined in Figure 4.3:

1. In Phase 1, a single copy of each n exists.

2. By the end of Phase 2, a second copy n' of each n has been created in M'. No insert or delete operations are allowed in M_{right} or M', so each n' cannot diverge from n.

3. In Phase 3, each original n is removed from the reachable set of Cell nodes. Because searches might already be traversing M_{right}, each n remains identical to the corresponding n'; insert and delete operations are still inhibited.

4. At the end of Phase 3, the `invalid` bit of each n is set to invalidate that n. Once $\forall n$ have been invalidated, each n' is the only remaining copy of that node, so insert and delete operations may resume in M and in the now-empty M_{right}.

5. The emptied nodes in M_{right} may be later reused; such a reused node n will first be given a new $minkey(n)$ and $maxkey(n)$ before the `invalid` bit is cleared. A slow search arriving at a reused node will no longer be able to use the `invalid` bit as evidence that backtracking is necessary, but the new range defined by $minkey(n)$ and $maxkey(n)$ will allow the search to proceed correctly.

Therefore, at each phase of the meganode search, the presence or absence of any key x in the Cell tree is unambiguous. □

The following theorem modifies Proposition 4.1, and because it shows that the three invariants from the proposition hold, Theorem 3.2 also holds.

Theorem 4.2. *Insert, delete, and search operations remain serializable during a meganode split, as outlined in Figure 4.3. Insert and delete operations are blocked for the duration of 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 serializable, by Theorem 3.2.*

Proof. We repeat the invariants from Proposition 4.1, and specifically address the changes necessary to prove each invariant in the face of concurrent dictionary operations and meganode splits.

Claim 1. If key x is in node n, then x is in the keyset of n.

By Lemma 4.2, all keys x in any node n in X_{right} (or in the new meganode) are in the keyset of n during a meganode split. The meganode split operation temporarily suspends write operations in X_{left}, but does not modify how they function, so insert and delete operations that continue in X_{left} proceed normally and continue to follow the invariants in Theorem 4.1.

Claim 2. The leaf nodes in Cell partition the KeySpace, or are exact, read-only copies of other nodes.

A meganode split copies a portion of an existing meganode to a new meganode, preserving the order and number of leaf-level nodes (which from Theorem 4.1 always partition the KeySpace). As outlined in Lemma 4.2, the only nodes that violate Claim 2 in Proposition 4.1 are those that are exact, read-only copies of existing nodes; these copies are eliminated by the time the meganode split completes. As Lemma 4.2 shows, the correctness of searches is maintained throughout the meganode split, while insert and delete operations are inhibited in the affected regions.

Claim 3. If the search for key x is at node n, then x is in $keyset(n)$ and n is a leaf node, or there is a path from n to a node m where $x \in keyset(m)$ and every path from n to m has x in its outset.

Every node n in X_{right} follows Claim 3 in Proposition 4.1 until it is invalidated. The only modification to the copies n' of the nodes n placed into the new meganode is to update each $rightlink(n')$ to point to the copy of the target node of n's rightlink. Until n is invalidated and n' is released from read-only to read-write (at which point it follows Claim 3 in Proposition 4.1), $x \in keyset(n')$ iff $x \in keyset(n)$ by Lemma 4.2. Therefore, because Claim 3 holds in Proposition 4.1, it holds here as well.

Conclusion. Any search through a Cell store for x, if it terminates, will terminate in the correct place, the node $n \mid x \in keyset(n)$, even in the face of meganode splits.

During meganode split phases 1 and 2, the Conclusion from Proposition 4.1 remains valid. During phase 3, the nodes in the old X_{right} are invalidated and the new X_{right} is linked into the tree. Searches that reach the old nodes after invalidation will restart. Searches that reach the new nodes will terminate normally. Searches that reach the old nodes after reuse (as in Claim 1) will not terminate, because the keyset defined by the min key and max key of the reused node will indicate that the node is not the correct termination point for the search. □

Therefore, because the invariants from Proposition 4.1 hold, and Theorem 4.1 remains correct, Cell operations remain serializable during a meganode split.

4.3.3 Concurrency in Key-Value Extents

We verify reads from the extents region using the technique proposed in Pilaf [54] of storing a CRC over the key-value entry in the corresponding leaf node pointer. After performing an RDMA in the extents 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. If the leaf node or the key-value extents were modified between the RDMA read of the leaf and the extents, the CRC will not match, and the client backtracks to re-read the leaf node.

By following Pilaf's approach to self-verifying pointers, the consistency guarantee over Cell's key-value extents also resembles the approach used for Merkle Trees [52] (see Section 3.3). However, in Cell's current implementation, parent B-tree nodes do not provide information to check the integrity of child nodes. If Cell's tree was built on a Merkle design, any modification to any node (including key updates, insertions, and deletions) would require updating every node in that branch of the tree. A distributed store design in which a single store-wide root must be modified for every write operation is necessarily impractical and cannot scale. Cell's approach uses the head and tail version number to guarantee that a node's contents are consistent, and the tuple of (`valid` bit, level, min key, maxkey) within a node give a client enough information to determine whether its traversal for some key K remains correct.

4.4 Combining Client-Side and Server-Side Search

Cell's hierarchical B-tree design allows for both server-side and RDMA searches. When servers are under low load, server-side searches are preferred for better overall resource efficiency. However, clients should switch to using RDMA searches when servers become overloaded. How should clients decide which search method to use dynamically?

To answer this question, we model the system using basic queuing theory [22]. 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 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) [21, 75]. 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:

$$\frac{q_s}{T_s} < m \times \frac{q_r}{T_r} \tag{4.1}$$

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 4.1 gives us the final search choice strategy.

To determine inequality 4.1, we need to estimate various terms. For a 64MB meganode with uncompressed 1KB nodes, we initially set $m = 5$; 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. Additionally, we apply the following approaches to improve the performance of the locality selector:

- *Coping with transient network conditions:* The measured latency l_s and l_r might be affected by transient network conditions that we do not wish to model. We cope with this in two ways. First, we use a moving average estimate of l_s and l_r: clients keeps 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 which seems too long or too short. Specifically, we exclude 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:* With a small probability (e.g., 1%), we choose the method estimated to be worse for a given search, to ensure our estimates for both methods are kept fresh. 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 HCAs and under various network and CPU loads.

4.5 Failure Recovery and Transactions

Distributed systems need to be resilient to application crashes and machine failures. 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 accessible from the network and replicated on multiple servers for good reliability and availability, e.g. Amazon's EBS or HDFS. Our prototype implementation does not yet use networked log storage, but 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 server logging is largely straightforward except in one scenario. 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 checks for orphaned meganodes upon finding unfinished split in the log. Node splits are handled similarly.

Transactions. Although we do not evaluate it here, Cell also includes support for distributed transactions using Optimistic Concurrency Control (OCC). The OCC protocol used is based on prior work [6, 2]. Clients read from the B-link tree but buffer writes locally during transaction execution. At commit time, clients perform a two phase locking protocol. The first phase locks items in the transaction's writeset and validates that items in its readset have not changed; the second phase writes items to the B-tree and unlocks. Servers log the outcome of the first phase durably in order to recover from server failure during 2PC. We make one optimization to this standard protocol: if a client detects a locked item from a

B-tree read, indicating that a conflicting transaction is in the process of committing, the client aborts its transaction early.

4.6 Summary

By selectively relaxing locality between data and computation, Cell can use server-side searches to complete B-tree traversals in few network round trips, and maintain low latency with client-side searches when servers' CPUs are saturated. Cell exemplifies a system that follows Insight 2 to achieve the best performance and load balancing with modest server CPU resources. Cell clients quickly and correctly switch between server-side and client-side operations as load shifts, and as the following two chapters will demonstrate, Infiniband-connected clusters of 1 to many Cell servers achieve high throughput on few CPU cores.

5
IMPLEMENTATION

Based on the design of Pilaf in Chapter 3 and of Cell in Chapter 4, we created prototypes of the two systems. Both Pilaf and Cell are implemented in C++. Both use Infiniband via the `libibverbs` library from the OpenFabrics Alliance, which allows user-space processes to use Infiniband's RDMA and message-passing primitive using functions called *verbs*. Although we investigated the Unreliable Connection (UC) transport that Infiniband provides [32], both Pilaf and Cell use Reliable Connection (RC), as it is the only option for performing both RDMA reads and Send/Recv verbs on the same connection. We use Send/Recv verbs to create a simple RPC layer out of messaging passing, used for both client-server and server-server operations. Clients (and in Cell, servers) also perform RDMA reads to fetch structural and extents data from servers.

5.1 General Techniques

Since Pilaf and Cell were developed on similar design principles for similar datacenter interconnects, they share several techniques. For instance, the implementation of consistent RDMA-readable data segments in Cell is informed by Pilaf's self-verifying data structures. The two systems have similar durable logging to SSDs, and manage large extents regions for key-value pairs the same way.

RDMA-Friendly Extents: Cell and Pilaf servers maintain key-value extents. In each case, a server must register a region of memory and give clients the registration key for that memory before clients can perform RDMA on the region. This process is relatively expensive and should be made infrequent. Therefore, Pilaf and Cell servers allocate and register a large contiguous address space for the key-value extents region. We ported the `mem5` memory management unit from SQLite to C++ to "malloc" and "free" strings in the key-value extents. Whenever a Pilaf server's extents region becomes full, the server resets all existing connections, expands the extents, and then allows clients to re-connect and obtain new registration keys. As with hash table resizing, we expect and observe extents resizing to be an infrequent event. Indeed, in Cell, we simply preallocate the extents to some desired size, and assume that additional servers can be added to the Cell cluster if it exhausts aggregate available memory. Dynamic resizing of

the Cell cluster is not currently implemented.

Self-Verifying Data Structures: CRC64 is used as the checksum scheme for most of the self-verifying data structures. CRCs are not effective for cryptographic verification. Instead, they were originally intended to detect random errors, making them a good fit for our application. The ideal n-bit CRC will fail to detect 1 in 2^n message collisions. Although 32-bit CRC is popular (e.g. for Ethernet and SATA checksums), we believe that CRC32 is insufficient for Pilaf and Cell. Every Pilaf put incurs two CRC updates, one on the hash table entry and one on the key-value string; every Cellput requires updating the key-value string's CRC. As will be shown in Section 6.2.2, Pilaf can process 663K puts per second on one core, and Cell can process 538K puts per second on two cores. Therefore, up to 1.326 million CRCs may be calculated per second per core in the two systems. Since each CRC32 incurs a collision with probability 1 in 2^{32}, we expect a collision at most once every 3239 seconds (54 minutes). We find this rate to be unacceptably high. Using CRC64, we can expect a collision at most once every $1.35 * 10^{13}$ seconds or 428 millennia.

CRC64 is fast. Our implementation consumes about a dozen CPU cycles for each checksummed byte, and incurs the same overhead as CRC32 when running on 64-bit CPUs. To further reduce server's CPUs, we offload CRC calculation to clients. The client is responsible for calculating the CRC to be used in the self-verifying pointer to a key-value pair when it performs a put operation: in Pilaf, the server need only calculate the (short) CRC over the hash table entry; in Cell, no additional CRC is computed.

Cell's small (RDMA-readable) nodes are not protected by a CRC, as recomputing the CRC of 1KB nodes on every insertion and deletion was deemed unnecessarily expensive. Instead, a version number is prepended and appended to each node. Nodes are only modified with the lowest bit of each version number set, and both versions are incremented before and after writes. Figure 4.4 and Section 4.2 provide more details on how this strategy guarantees that no incomplete modifications can be overlooked by RDMA reads.

Logging: By default, the Pilaf and Cell servers asynchronously log write operations such as put and delete to the local disk, similar to the logging facility in other key-value stores including Redis [62], Masstree [48] and LevelDB [18]. Using a single solid state disk, both Pilaf and Cell can log operations as fast as they can process them. Pilaf is able to log at least 663K writes per second (its peak single-core

put throughput) if the average key-value size is smaller than 500 bytes. Because of the more complex structural updates needed to perform B-tree insertions, Cell necessarily completes fewer operations per second per core. It is able to log at least 538K small key-value operations per second (its two-core 100% put throughput). Should a higher logging capacity or additional active server cores be required, multiple SSDs can be used.

5.2 Pilaf Implementation

Pilaf is written in ~6,000 lines of C++. It includes a custom-built Infiniband library to simplify connection management, a server and client built on top of this library, and the requisite tooling to evaluate its performance. The Pilaf server continuously polls the network card for new events, including the reception of Verb messages or the completion of Verb sends. Since Pilaf is able to saturate the network card's performance using a single thread, our server implementation uses the same polling thread to process puts as well. The Pilaf client is also single-threaded, issuing RDMA or Verb requests and waiting for the response before commencing the next operation.

Distribution and Sharding: As Pilaf is a distributed in-memory key-value store, clients must be able to connect simultaneously to an arbitrary set of Pilaf servers. We use *deterministic client-side sharding* to distribute keys among multiple servers, a well-established practice to eliminate the need for a centralized index [59]. Given that each client has an identical list of available servers, every client will query the same server for any given key.

5.3 Cell Implementation

Cell is implemented in ~16,000 lines of C++. It includes an improved, thread-safe version of Pilaf's Infiniband library. The single-threaded Cell server continuously polls the Infiniband NIC for new events, including the reception of incoming messages or the completion of outgoing messages. The polling thread synchronously processes RPC requests and performs server-server interactions for meganode splits. Like HERD [32], we run multiple server processes per machine in order to take advantage of multiple CPU cores. The Cell client also polls for network events that indicate the completion of outgoing requests,

reception of incoming replies, and completion of RDMA reads. The most recent iteration of the Cell client is multi-threaded: one thread handles network polling, delivering incoming messages and received RDMA read data to client threads for further processing. To achieve good parallelism, the client also supports pipelined operations in each thread to keep multiple key-value operations outstanding at any given time.

Client-side searches use RDMA reads to repeatedly fetch 1KB nodes to traverse a meganode. Clients cache B-link tree nodes to accelerate future traversals, maintaining an LRU cache of up to 128MB of 1KB nodes. We choose only to cache nodes at least four levels above the leaf level of the tree to minimize churn and maximize hits. 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. Symmetrically to RDMA, we maintain an LRU cache of up to 4K server-side traversal paths leading to the leaf-level meganodes, each specifying the node at the target level known to hold a given key range.

Servers store B-link nodes in one dedicated memory pool and key-value extents in a second pool. Because the entire memory region over which RDMA will be performed must be known when clients connect to servers, Cell servers pre-allocate these large memory pools, then manage the memory allocation within these pools by itself. In our current prototype implementation, the node pool has very simple allocation management because allocated nodes have a fixed size and they are not freed. For the extents pool that hold actual key-value tuples, we use a version of SQLite's `mem5` memory manager ported to C++, optimized, and made compatible with memory pools larger than 4GB. We support `hugetlbfs` backing for these large memory pools to reduce RDMA read latency [16].

Node size: 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. The downside of smaller nodes is that tree depth increases, but the low RTT of RDMA reads compensates for this. Our B-link tree design means that halving the node size halves the amount of unused data transferred on average while adding a single level to the tree. RDMA-based traversals do incur a significant bandwidth amplification over server-side searches, on the order of $4{\cdot}1KB/256bytes = 16\times$ with caching enabled. However, because 1KB RDMA reads are on the boundary between the maximum ops/sec our HCAs can perform and bandwidth saturation (demonstrated in Figure 2.3), smaller nodes could not be read more quickly. Using 512-byte

reads to fetch 512-byte small nodes would add 4 levels (33%) to a 10^{15}-key tree while only enabling 16% more node fetches/HCA/sec, increasing traversal times while further amplifying bandwidth usage.

6
EVALUATION

The new design approaches this thesis advocates for RDMA-enabled distributed systems appear qualitatively effective from analysis. To prove that these designs are quantitatively successful, we evaluate the performance of Pilaf and Cell on Infiniband-connected clusters. Our results are aligned with five important high-level questions:

- **Are systems with relaxed locality fast?** Pilaf achieves high performance: its peak throughput reaches ∼1.3 million ops/sec. The end-to-end operation latency is very low with a 90-percentile latency of ∼30μs. Cell is also fast, outperforming alternate implementations that do not relax locality by using a combination of client-side and server-side processing when servers are bottlenecked on the CPU. It can sustain 5.5 million search ops/sec using 16 Cell servers, each of which uses 2 cores.

- **Can selectively relaxed locality save CPUs?** Pilaf is CPU-efficient. Even when running on a *single* CPU core, Pilaf is able to saturate the network hardware's capacity to achieve 1.3 million ops/sec. By comparison, Memcached and Redis achieve less than 60K ops/sec per CPU core, so they require at least 20× the CPU resource to match Pilaf's performance. Cell can provide high throughput and low latency when servers are provisioned for a mean load. Using only 2 CPU cores in a server, Cell can perform 545K get ops/sec, equivalent to the 4-core performance of the server-side search scheme.

- **Do these technique scale?** Cell and Pilaf scale well across many servers and many clients. We observe a near-linear speedup as we increase the number of clients and servers for both systems, bounded by the maximum throughput of the Infiniband interconnect and in some cases the servers' CPUs. Cell achieves 5.57M search ops/sec on 32 cores across 16 servers.

- **Do self-verifying data structures effectively detect memory races?** No RDMA-fetched memory was incorrectly deemed valid in all these experiments. Self-verifying data structures are effective at detecting read-write races between the clients' RDMA operations and the server's local memory accesses. Self-verifying object techniques can be extended to arbitrary data structures, allowing clients to correctly detect invalid reads. They provide a framework on which clients can also detect moved or deleted items or incorrect traversals through the structure and correct themselves

accordingly. Pilaf and Cell exploit three different variations on self-verifying objects that succeed in detecting read-write races and structural changes.

- **Can selectively relaxed locality improve load balancing and handle load spikes?** Cell minimizes latency and maximize throughput regardless of load. With caching enabled and 2 cores per server machine running Cell servers, selectively relaxed locality yields root-to-extents latencies as low as $40\mu s$ for a 320M-key tree under moderate load, and throughput up to 492K ops/sec/server with 4 client processes/server. More importantly, the locality selector allows clients to maintain low latency in the face of transient load spikes on servers provisioned below the peak load. When a 5-second load spike at $3\times$ the nominal request rate occurs, Cell clients shift the bulk of searches to client-side operations within $200ms$ to maintain low latency.

This chapter begins with the experimental setups on which Pilaf and Cell were tested in Section 6.1, to make comparisons to other systems (and to each other) easier to draw. We present the performance of Pilaf and Cell on a single core of a single server in Section 6.2 to provide context for our multi-core and multi-server scaleout experiments. The techniques used in Pilaf and Cell scale well to many servers and server CPU cores, demonstrated through Cell's scaleout performance in Section 6.3. The probability of conflicts in an in-memory store are measured in Section 6.4, and the effectiveness of Cell's locality selector is evaluated in Section 6.5. To compare to other popular key-value stores and sorted stores, we present the performance of Cell and Pilaf against other systems and with realistic macrobenchmark workloads in Section 6.6. Finally, Section 6.7 details the effects of durable logging on each system.

6.1 Experimental setup

Three Infiniband-connected commodity clusters were available for testing: a local cluster, and two PRObE clusters, Nome and Susitna. Pilaf was tested on our local, 10-machine cluster, using up to 4 machines for servers and 6 machines for clients. For Cell, we ran experiments on the larger Nome PRObE cluster, using up to 16 of the 256 nodes for Cell servers, and up to 48 additional nodes for clients. We performed microbenchmarks for raw Infiniband operations and Cell on our local cluster, the Nome PRObE cluster, and the Nome Susitna cluster.

6.1.1 Hardware

The three clusters we used have similar Infiniband interconnects but various CPU and memory configurations:

Local "Beaker" cluster: Our local machines, used for Pilaf and raw Infiniband testing, each have two AMD or Intel processors and 32GB of memory. The machines used for servers have two AMD CPUs with 8 physical cores each, and the machines used for clients have two Intel CPUs with 4 physical cores each. Each machine is also equipped with a Mellanox ConnectX VPI DDR (20 Gbps) Infiniband HCA as well as an Intel 1 Gbps Ethernet adapter. The machines run Ubuntu 12.10 with the OFED 3.2 Infiniband driver.

Nome PRObE cluster: Our Cell and raw Infiniband scaleout tests were performed on a subset of the 256 machines in the shared Nome PRObE cluster. Each machine is equipped with 4 quad-core AMD processors and 32GB of memory, as well as a Mellanox ConnectX EN DDR (20Gbps) Infiniband HCA and two Intel gigabit Ethernet adapters. Tests were run on CentOS 6.5 with the OFED 2.4 Infiniband drivers.

Susitna PRObE cluster: For some raw Infiniband tests and Cell microbenchmarks, we used up to 16 machines in the shared Susitna PRObE cluster. Each node has four 16-core AMD processors and 128GB of RAM. Each machine also has a Mellanox ConnectX VPI FDR (14Gbps) Infiniband HCA and a Mellanox ConnectX-3 QDR (40Gbps) RoCE (RDMA over Converged Ethernet) HCA. Our tests ran on CentOS 6.3 with the OFED 2.4 Infiniband drivers.

6.1.2 Configuration

Experimental configurations were designed to emulate realistic datacenter scenarios, albeit on a smaller scale than the typical datacenter.

Pilaf: For each experiment, we run a server process on one physical machine, while the clients are distributed among the remaining machines to saturate the server. Because Pilaf's servers are entirely independent and do not perform any inter-server communication, we do not present results for multi-server

clusters. By default, we restrict the server process to run on one CPU core. For Ethernet experiments, we configure the kernel's network interrupt processing to trigger on the same core used by the server process.

We disable Pilaf's asynchronous logging in the experiments. With logging turned on and a 100% put workload, Pilaf incurs no measurable reduction in achieved throughput for key-value sizes less than 500 bytes. With larger operations, the I/O bandwidth of the server's single local SSD becomes the bottleneck.

Cell: For each experiment, we use a cluster of server machines separate from client machines. Unless otherwise mentioned, we use four servers and utilize two cores per server (by running two server processes per machine). We devote the remaining machines to running clients. The cluster is pre-populated with 20M key-value pairs per server machine, unless otherwise indicated.

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 due to other sources of Infiniband latency at scale [16].

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.

With durable logging enabled and a 100% put workload on 2 cores per server, Cell incurs no measurable reduction in achieved throughput for key-value sizes below 575 bytes. With larger operations, the I/O bandwidth of each of our local cluster's server's single local SSD becomes the bottleneck for workloads with large ratios of write to read operations. As Nome's machines are not equipped with SSDs, we disable Cell's asynchronous logging in our experiments.

6.1.3 Workloads

Workloads are also designed to approximate real-world distributed store work, including the size of the data elements and the distribution of operations performed.

Pilaf: We use the YCSB [10] benchmark to generate our workloads. YCSB constructs key-value pairs with variable key and value lengths, modeled on the statistical properties of real-world workloads. Furthermore, with YCSB, the keys being accessed follow a long-tailed zipf distribution. The original YCSB

software is written in Java. We ported it to C so that fewer client machines are required to saturate the server, and implement lightweight clients in C for Pilaf, Memcached, and Redis to use the YCSB workload files.

In all experiments, we vary the size of the value string from 16 to 4096 bytes while keeping the average key size at 23 bytes, the default value in YCSB. We use two mixed workloads, one consisting of 10% puts and 90% gets, the other 50% puts and 50% gets. Since Facebook has reported that most of their Memcached deployments are read-heavy [4], our mixed workloads give reasonable representations of real workloads. For each test, we apportion operations across "popular" keys following a Zipfian distribution, calculated by YCSB.

We compare Pilaf against Memcached [17] and Redis [62] (with logging disabled). We also compare Pilaf to an alternative implementation of itself, which we refer to as Pilaf-VO (short for Pilaf using Verb messages Only). In Pilaf-VO, clients send all operations (including gets) to the server for processing via Verb messages. The performance gap between Pilaf and Pilaf-VO demonstrates the importance of relaxing locality.

Cell: To test search, get, and put operations, Cell generates 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. We also test mixes of get and put operations from 100% get to 100% put.

6.2 Microbenchmarks

The first question to be answered is "**Are systems with relaxed locality fast?**" This section demonstrates that operating on a single machine, a single core, or a single section of a data structure, Pilaf and Cell surpass the latency and throughput of schemes that maintain strict locality. A common fallacy is to demonstrate near-linear scaling from a suboptimal one-core base case [49], so we show that even the results for a single CPU core are optimal. This section presents the throughput and latency of individual Pilaf get and put operations on a single server utilizing a single CPU core. It 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.

6.2.1 Raw Infiniband Operations

We measure the throughput and latency of RDMA, two-way Verb ping-pongs, IPoIB, and 1Gbps Ethernet for a range of message sizes in Section 2.2. Up to 2.44M RDMA ops/sec and 668K Verb message ops/sec are possible for small messages with one server core, while larger operations saturate the HCA at the signaling throughput of 16Gbps. We also measure the throughput and latency of 1KB RDMA reads and 128-byte two-way Verb ping-pongs on clusters of 1 to 24 servers, utilizing 1 CPU core per machine; this data is presented in Table 2.3 (Chapter 2). Because very little per-message processing is performed on the servers, we do not observe a CPU bottleneck. We varied the client count to search the throughput-latency space for RDMA reads, Verb messaging, and simultaneous use of the two. Because latency rises with no additional gain of throughput past saturation, and we cover the client count space sparsely, we report the throughput within 5% of the maximum measured throughput with the lowest latency. As discussed in Chapter 2, we observe a performance cliff with hundreds of active Infiniband connections per HCA on Mellanox ConnectX hardware [16]. We anticipate that future Infiniband networking hardware will repair this limitation.

6.2.2 Pilaf Microbenchmarks

Pilaf was tested on the local "Beaker" cluster, using 1 machine for a single-core Pilaf server and 6 machines with up to 8 Pilaf clients each. get and put throughput and latency were measured over a range of value sizes from 16 bytes to 4KB.

Throughput: Figure 6.1 shows Pilaf's peak operation throughput, achieved with 40 concurrent clients processes on 6 client machines. Pilaf can perform ~1.28 million get and 663K put operations per second for small key-values. Of note is that Pilaf's high throughput is achieved using a single CPU core which saturate the Infiniband card's performance for in most cases.

get operations via RDMA impose zero CPU overhead on the server. Furthermore, get operations also have the highest throughput. As shown in Table 2.2 (Chapter 2), the network card's potential RDMA throughput is much higher than that of Verb messages, especially for small messages. In particular, the card can satisfy 2.45 million RDMA reads per second for small reads. Since each get requires at least two RDMA reads, the overall throughput is approximately half of the raw RDMA throughput at 1.28 mil-

lion gets/sec. By contrast, the peak verb throughput is 667K request/reply pairs/sec for small messages, resulting in 667K ops/sec for puts.

For larger key-value pairs, the throughputs of get and put converge as they both approach the network bandwidth. For example, for 4096-byte key-values, a one-core Pilaf server consumes 11.7Gbps of the 16Gbps data bandwidth supported by the network card. Interestingly, we find that when processing puts with large values, the Pilaf server becomes CPU-bound on a single core. Specifically, for 1024-byte value size, Pilaf achieves 75.4% of its network-bound put throughput (500K ops/sec) with one core and 100% (663K ops/sec) with two cores.

We also measure the throughput of Pilaf-VO's get operation, which is processed by the server using Verb messages instead of by the client using RDMA. As Figure 6.1 shows, the throughput of performing gets using Verb messages is similar to that of puts and is much smaller than the throughput of gets done via RDMA for small key-value pairs. The disparity between the pure get and put throughputs are due to the slightly higher server CPU cost of storing large key-value pairs.

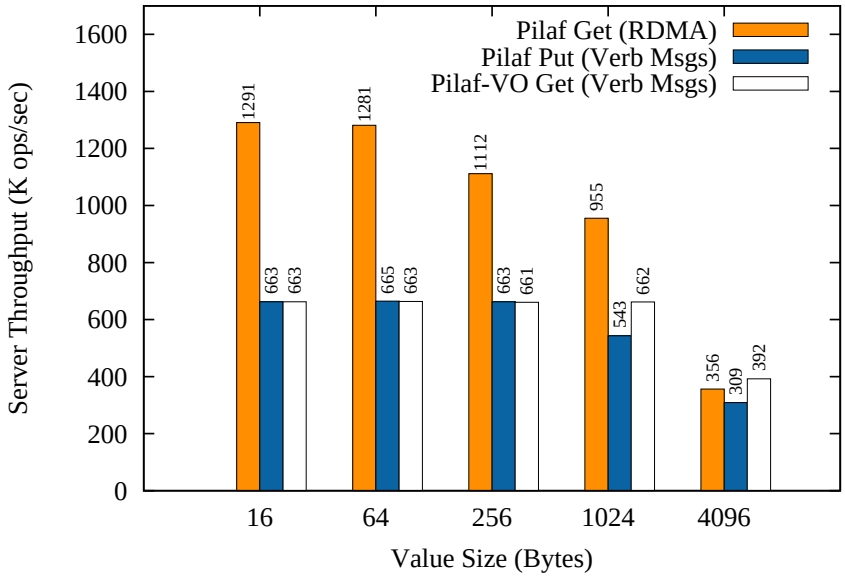


Figure 6.1: Server throughput for put and get operations as the average value length is increased. All tests are performed with 40 connected clients.

Latency: Figure 6.2 shows the latency of Pilaf get and put operations with 10 concurrent clients. With 10 concurrent clients performing operations as fast as possible, queuing effects are minimized. With 40 or more clients, the latency is mostly determined by queuing effects and thus is much higher. With a single client (not shown in the figure), the latency of get is slightly more than 2 RDMA round trips and is twice the latency of put. With more clients and thus more load, we found that the RDMA latency scales better than that of verb messages. For small gets, the average latency is 12μs, while small puts take around 15μs. For large key-values, the latencies of get and put are similar and both bounded by the packet transmission time.

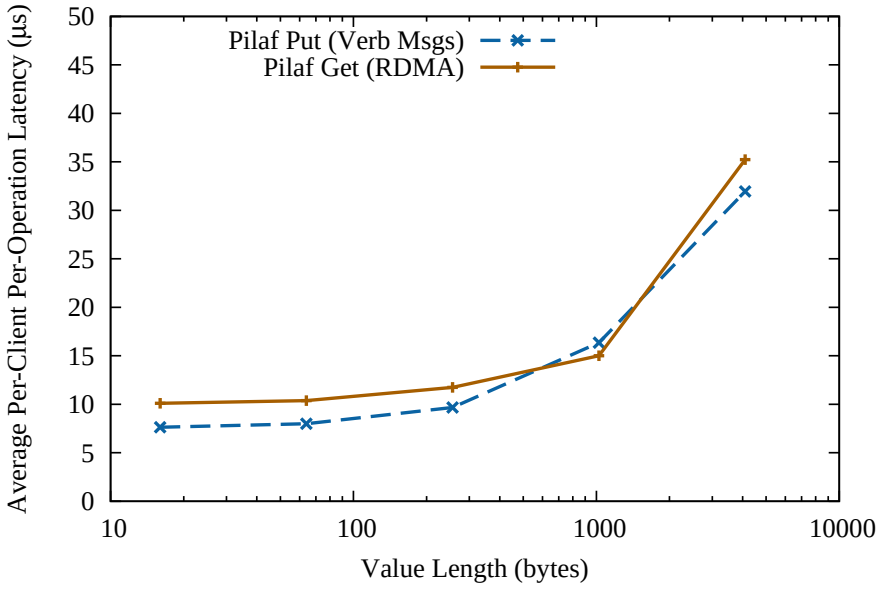


Figure 6.2: Average operation latency for put and get operations as the average value size increases. All tests are performed with 10 connected clients; though not pictured, we observe a linear relationship between the number of connected clients and latency due to queuing effects.

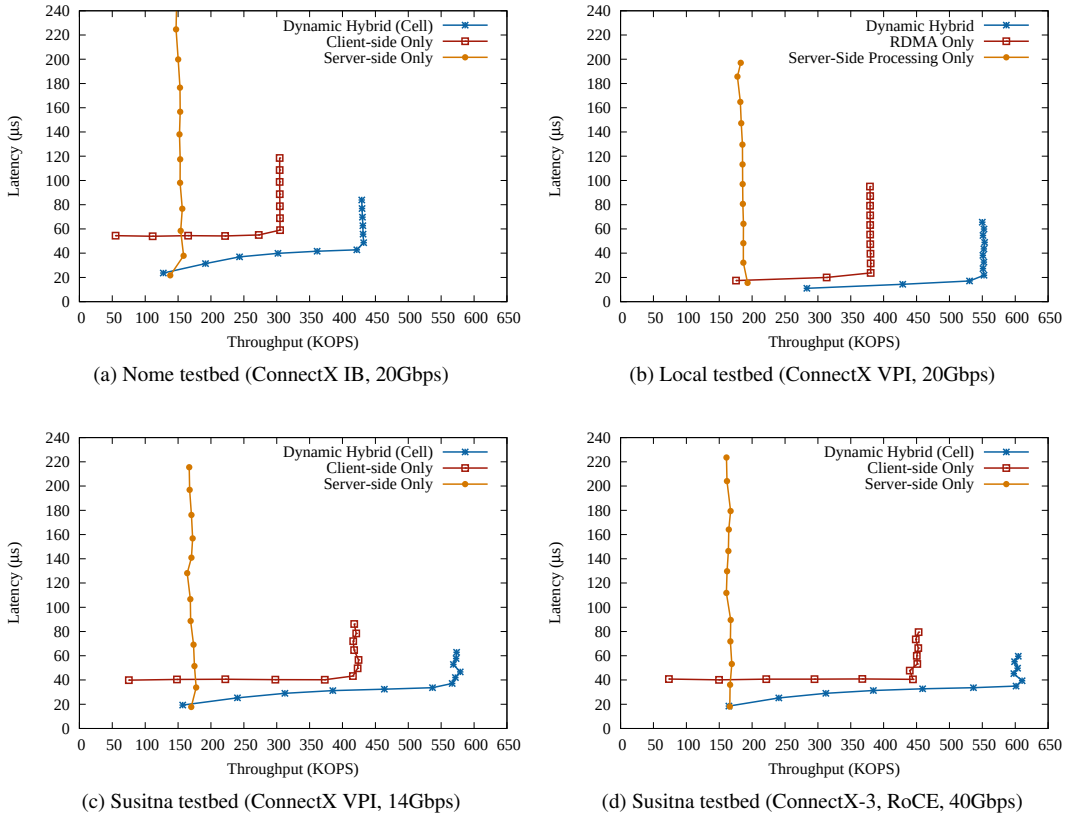
6.2.3 Cell Microbenchmarks

In Chapter 2, we explored Infiniband's performance on two clusters using two different types of Mellanox Infiniband HCAs. In this section, we compare those results to Cell's performance when traversing a single meganode on 1 server CPU core with a small number of clients. We examine the performance of client-side only searches, server-side only searches, and Cell's hybrid approach. Figure 6.3 demonstrates the

throughput-latency relationship as we increase the number of clients from 1 to 12 processes, distributed across 6 client machines. We present performance from three different clusters (Nome, Susitna, and our local Beaker cluster); Cell experiments throughout this section are run on Nome. As with other Cell experiments, each client executes three separate traversal threads, but unlike other experiments, all client-side caching is disabled. With the server performing B-tree searches, 1 server CPU core is no longer sufficient to saturate the Nome NICs. The server-side search's peak throughput is 100K searches/sec, and the bottleneck is the single server CPU core. Client-side only searches peak at 250K searches/sec; because 4 round trips are required to traverse the meganode, this is equivalent to 1.00M 1KB RDMA reads per second. Table 2.3 suggests that the peak throughput for Nome's NICs is 1.04M 1KB RDMA reads/sec, so the NIC's operation throughput is saturated. Searches utilizing selective locality improve considerably on each individual method, peaking at close to the aggregate throughput of the two methods (~400K searches/sec).

The second question posed in this chapter is "**Can selectively relaxed locality save CPUs?**" In a system with many CPU cores available, a server can saturate any network hardware or medium with low-latency server-side searches. However, provisioning resources in a shared environment requires selecting the minimal CPU resources necessary to handle an expected level of workload demand. Cell dynamically selects between server-side searches and RDMA searches to ensure that latency remains low as the load on servers changes, and allows a Cell storage cluster to satisfy transient peak usage without over-provisioning for mean usage. We benchmark Cell's ability to scale across many cores in one server machine. In this test, we used 1 to 8 cores in a single server to host single-threaded Cell servers. Each server is part of the same Cell cluster and holds one five-level meganode. Clients perform repeated search operations, traversing the full height of a single meganode via server-side or client-side search; caching is disabled. Experiments are run with server-side search only, with Cell's hybrid approach, and with the fixed ratio of server-side and client-side searches that yields the highest throughput for that server CPU core count. Our results are presented in Figure 6.4. Quantitatively, Cell's hybrid approach yields more than 2 cores of "free" performance per HCA per machine. In other words, to get Cell's 1-core performance (436K ops/sec), server-side search requires more than 3 cores, while to get Cell's 4-core performance (778K ops/sec), server-side search needs at least 6 cores.

With baselines for Pilaf's and Cell's performance on a single server established, we scale the systems across many server machines and cores.



(a) Nome testbed (ConnectX IB, 20Gbps)

(b) Local testbed (ConnectX VPI, 20Gbps)

(c) Susitna testbed (ConnectX VPI, 14Gbps)

(d) Susitna testbed (ConnectX-3, RoCE, 40Gbps)

Figure 6.3: Throughput and latency for pure server-side search, pure client-side search, and Cell's locality selector. The experiments involve 1 server serving one meganode, utilizing one core. B-tree caching is disabled. Performance on three different testbeds; Nome was used for almost all experiments presented in this chapter. Compare to the Infiniband microbenchmarks presented for each of these clusters in Table 2.3.

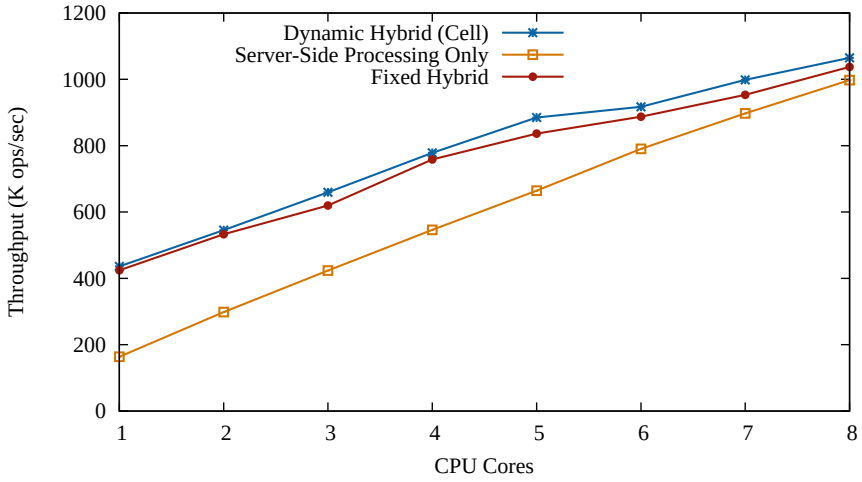


Figure 6.4: Cell throughput on 1 to 8 server CPU cores on a single machine, traversing a single meganode to perform a search operation. Server-side search and Cell's hybrid approach are both tested. In addition, the highest throughput for fixed ratios of server-side and client-side searches at each CPU core count is presented.

6.3 Scaling

Pilaf and Cell are designed as prototypes for datacenter-scale distributed systems architectures. They demonstrate features that are applicable to systems spread across dozens, hundreds, or even thousands of machine, so the third question in this chapter is "**Do these technique scale?**" We test how Cell scales to many server machines compared with the one-server, one-core microbenchmarks presented in the previous section. As discussed in Section 6.1, we tested Cell on a subset of the 256-machine Nome PRObE cluster. We find that the system achieves a near-linear speedup as additional server machines (and to some extent, server cores) are added, bounded by the limitations of the Infiniband hardware discussed here and in Chapter 2. We present our scaling results for Cell in this section.

We tested Pilaf on our local 10-machine cluster. Because it is a distributed hash table, requiring no inter-server communication under normal operating conditions, we do not present scaleout performance for Pilaf herein. As with Cell, the size of a Pilaf cluster is limited primarily by the number of active connections on each server. We also do not present results from scaling Pilaf to multiple cores per server. Pilaf's structure is sufficiently simple that verb put operations handled by a single core saturate the verb

operations/second capacity of our cluster's ConnectX VPI NICs.

6.3.1 Machine Scaleout

We investigate Cell's ability to scale by varying the number of server machines from 1 to 16. The experiments utilize two CPU cores per server; client caching is enabled. The size of the B-tree is scaled to the number of servers, at 20M tuples per server. We also use enough clients to saturate each set of Cell servers. Our biggest experiments consist of 59 machines (16 servers plus 43 clients) with 2560 client-server connections (80 3-threaded client processes to saturate the 16-server cluster, 2 cores per server). Our largest B-tree is 2 meganodes tall and stores 480M tuples.

Throughput: Figure 6.5 shows the search throughput of Cell as well as the server-side only alternative. The same throughput numbers are presented in tabular form as Table 6.1. Figure 6.5 and Table 6.1 demonstrate that Cell displays near-ideal scaling over additional servers. Cell's throughput increases $19.1\times$ from 1 server to 24 servers. Compared to the server-only approach, Cell's hybrid approach achieves 78% higher throughput at 24 servers.

Latency: The B-tree remains 2 meganode levels and 7 small node levels tall as the number of servers grows from 4 to 16. As such, the overall search latency is stable as the number of servers increases from 4 to 16. The median latency for the server-side only setup is $\sim 52\mu s$. The median latency for Cell is $\sim 40\mu s$. Since caching is enabled, these figures are better than Figure 6.3, even though 2 meganodes must be traversed for each search.

Servers	Messaging Thru	Hybrid Thru
1	260K ops	492K ops
2	526K ops	888K ops
4	986K ops	1626K ops
8	1848K ops	3025K ops
16	3129K ops	5572K ops

Table 6.1: Aggregate search throughput machine with varying number of servers. This is the same as Figure 6.5.

6.3.2 Multi-Machine Core Scaleout

The results from scaling Cell across 1 to 8 CPU cores on a single server machine were presented in Section 6.2.3. Cell's scaling performance across 1 to 8 cores each on 4 servers is also measured for com-

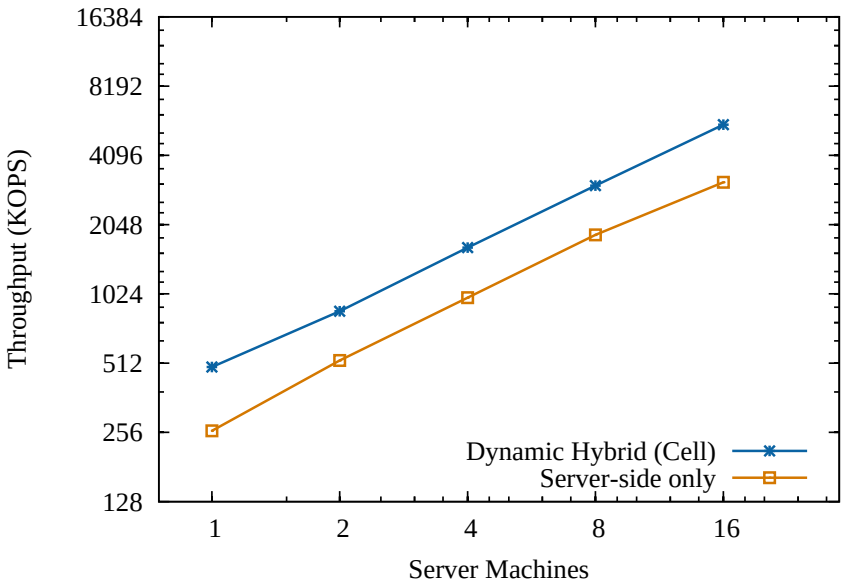


Figure 6.5: Throughput as the number of servers increases from 1 to 24. Each server uses 2 CPU cores. 4 client machines per server are used to saturate the servers. Caching is enabled.

parison. Table 6.2 shows the aggregate throughput for server-side searches only, Cell, and the fixed ratio of server-side to RDMA searches that yields the best throughput for each core count. The latency of hybrid operations remains consistently low; with 2 CPU cores, Cell searches average 17.7μs, dropping to 11.8μs with 4 cores per server. Our microbenchmarks indicate that the maximum Verb throughput Nome's Infiniband NICs support is 750KOPS for 128-byte messages and slightly higher for smaller messages, so we expect a theoretical maximum of 3.0M to 4.0M server-side searches per second. Table 6.2 indicates that Cell's hybrid scheme can extract 2.31M searches per second from 2 CPU cores on each of 4 server machines, double the 1.16M from server-side only searches at the same CPU count. Server-side searches are able to reach 2.95M searches per second on 6 cores. This 3$\times$ resource expenditure yields only 81% higher throughput compared with Cell on 2 cores per machines. As with the single-machine microbenchmarks, Cell provides roughly a 2-core advantage per HCA per server via client-side operations. Peak Cell performance on 6 to 8 cores on 4 machines occurs at 13 to 17 multi-threaded clients, or 102 to 112 connections per server machine. Due to Mellanox ConnectX HCAs' performance collapse with many connections, the 6 to 8 core results do not adequately represent the machines' peak throughput for those core counts.

6.4 Self-Verifying Data Structures: Collisions and Retries

The fourth question at the beginning of this evaluation asks, "**Do self-verifying data structures effectively detect memory races?**" Pilaf and Cell use the three techniques outlined in Chapters 3 and 4 to build self-verifying structures. Each technique guarantees that clients are able to detect and respond to memory access races between RDMA reads and server CPU writes. We expect such races to be rare in a normal workload, and we validate this assumption using Pilaf. To artificially vary the conflict rate, we inject the maximum achievable get and put loads, simultaneously reading and writing a varying number of unique key-value pairs. Therefore, the probability of races increases as the gets and puts are restricted to fewer and fewer unique keys. Notably, in this experiment (and all other experiments presented herein), no self-verifying data structure was incorrectly read as valid in the face of memory races.

Figure 6.6 shows the probability of detecting a read-write race as measured by the fraction of gets that need to be re-tried. The two lines in Figure 6.6 illustrate the probabilities of a retry due to a race when reading the hash table entry or when reading the key-value extent. As the figure shows, there is non-negligible race rate only when the hash table is extremely small. When the table contains more than 20K keys, the probability of racing is less than 0.01% even under peak put and get loads.

6.5 Locality Selector Performance

In Pilaf, clients use RDMA reads for all read-only operations, such as search and get. In Cell, a locality selector effectively chooses the correct search method to use under arbitrary network and server load conditions. As network and server load shift, the selector instantaneously chooses server-side or client-side searches for each operation. The selector is designed to ration server CPU resources by selecting the server queue (RDMA or server-side searches) that is least full. This has the practical effect of minimizing per-operation latency. We acknowledge that the locality selector used in Cell (the later work) could be used to enhance Pilaf, while cautioning that our experiments indicate that adding selectively-relaxed locality to a system like Pilaf that uses little CPU would only see benefits from a workload with an unrealistically high ratio of puts.

The fifth and final question at the beginning of this chapter asks, "**Can selectively relaxed locality improve load balancing and handle load spikes?**" We evaluate the performance of Cell's locality

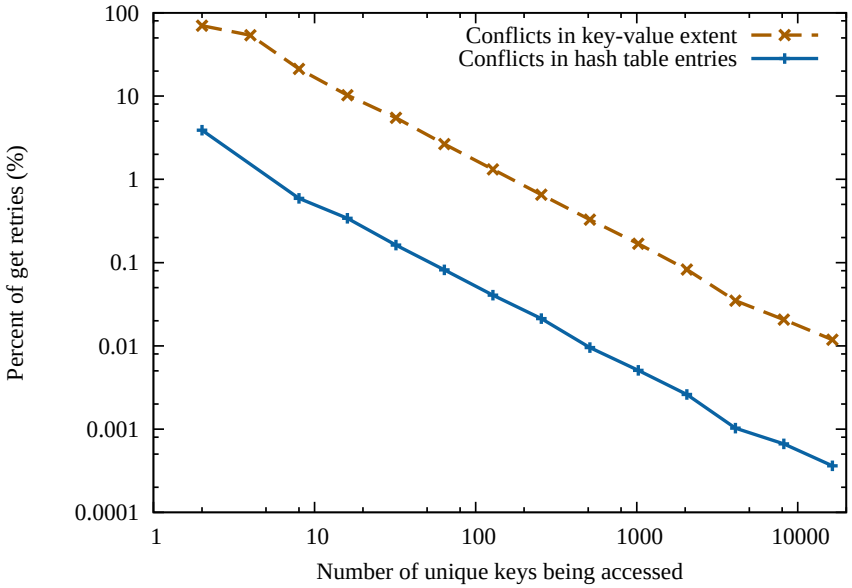


Figure 6.6: Percentage of re-reads of extents and hash table entries due to detected read-write races. We control the rate of conflicts by varying the number of unique keys being read or updated. The Pilaf server is operating under peak operation throughput.

selector by asking three more detailed questions about Cell:

How effectively does the locality selector use CPU resources? The comparison of Cell's locality selector and 100% server-side search in Table 6.2 shows that Cell is able to consistently match server-side search's throughput using at least 2 fewer CPU.

How accurately does Cell estimate search costs? Under a constant server load, a fixed ratio of server-side and client-side searches produces maximal throughput at minimal latency. However, as illustrated in Table 6.2, this ratio shifts dramatically as the available server resources change. Our experiments also reveal that the ratio of read to write operations in workloads further affects the correct ratio of server-side to client-side searches to use. Cell correctly picks near-optimal ratios 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 server's data structures, we observe Cell picking different ratios on each client that produce globally optimal throughput and latency. Figure 6.7 shows that in most cases, especially when

few resources are available, Cell meets or exceeds the throughput of a ratio hand-tuned to the current load conditions.

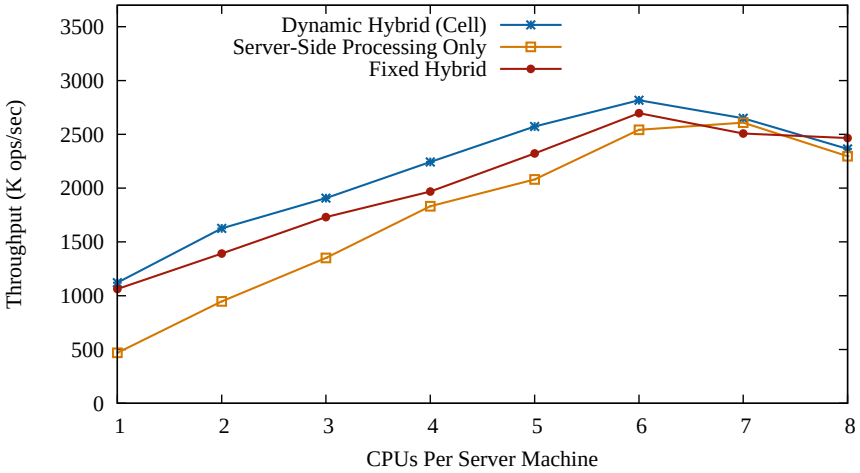


Figure 6.7: Throughput of a 4-server machine Cell cluster as more CPU cores are available, with Cell's dynamic hybrid approach and a manually tuned percentage of message-passing operations. Percentages indicate number of server-side searches; the remaining searches are client-side search.

CPU cores	Server-only	Cell	Fixed-Ratio
1	468K	1121K (37%)	1062K
2	947K	1626K (61%)	1392K
3	1350K	1907K (76%)	1731K
4	1832K	2269K (83%)	2178K
5	2184K	2668K (89%)	2258K
6	2523K	2832K (99%)	2676K
7	2947K	2810K (99%)	2494K
8	2720K	2757K (99%)	2468K

Table 6.2: Search throughput of 4 server machines utilizing 1 to 8 CPUs per server, performing 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.

When is the locality selector beneficial? 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. Figure 6.8 compares the application-facing latency of dynamic hybrid clients and server-side search-only clients when the request rate on a cluster of 4 servers increases unexpectedly for 5 seconds. In this test, application search requests arrive at each of 24 Cell clients every 75μs. For 5 seconds, the

load rises by 2.5× as an additional 36 clients begin injecting searches at the same rate. Figure 6.8 shows that Cell is able to very rapidly switch the majority of its searches to client-side traversals, maintaining a low median and 90th percentile latency compared to the server-side search. Note that Figure 6.8 has a logarithmic y-axis. 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.

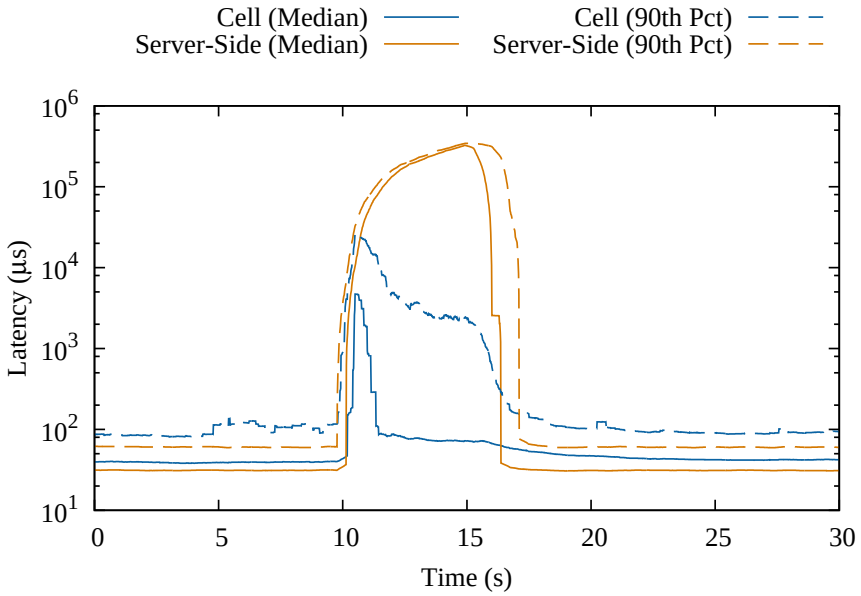


Figure 6.8: Application request latency over time during 5 seconds of transient load applied to a previously unsaturated cluster of 4 server machines hosting 8 Cell server processes. Medium and 90th percentile latencies are plotted on a logarithmic y-axis. These tests were performed with client pipelining disabled for simplicity.

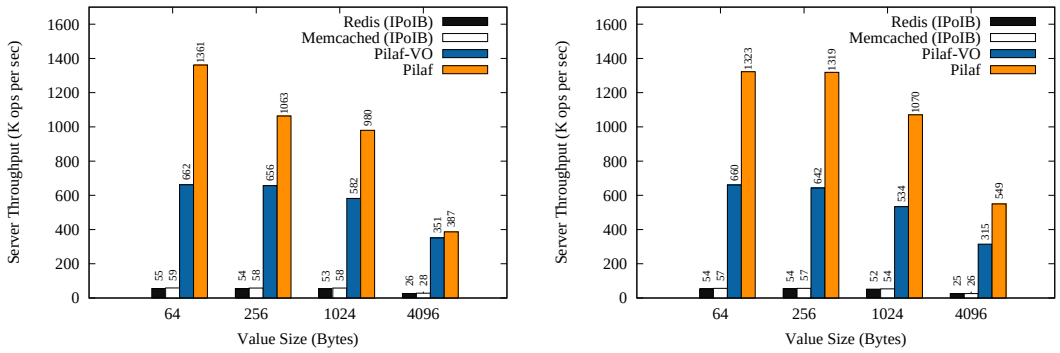
6.6 Realistic Workload Benchmarks

The microbenchmarks in Section 6.2 and scaleout tests in Section 6.3 show how Pilaf and Cell perform on single cores under controlled conditions, as well as with read-only workloads scaled across many servers and cores. Real-world performance is more difficult to measure, because real workloads are bursty, unpredictable, and often consist of a varying ratio of read and write operations. In this section, we compare the one-core Pilaf performance to the well-known Memcached and Redis key-value stores in Section

6.6.1. Because a B-tree store must perform more complicated operations than just search, Section 6.6.2 presents Cell's performance in workloads with real mixes of put and get operations.

6.6.1 Pilaf versus Memcached and Redis

We compare Pilaf to two existing popular key-value systems, Memcached [17] and Redis [62]. Both systems are widely deployed in the industry, including Facebook [4], YouTube [12], and Instagram [35]. Memcached is commonly used as a database query cache or a web cache to speed up the server's generation of a result web page and improve throughput. Low operation latency is vital in such a usage scenario: the faster the key-value cache can fulfill each request, the faster a page involving many cache lookups can be returned to the client. High throughput and low CPU overhead are also crucial, since these properties allow more clients can be served with fewer server resources. Because Memcached and Redis are written to use TCP sockets, we run them on our Infiniband network using IPoIB. It's important to note that we do not batch requests for any of the systems, unlike in [48].



(a) Peak throughput (90% gets, 10% puts)

(b) Peak throughput (50% gets, 50% puts)

Figure 6.9: Throughput achieved on a single CPU core for Pilaf, Pilaf-VO, Redis, and Memcached.

In our experiments, the peak throughput of each system is achieved when running 40 concurrent client processes. We use two mixed workloads, one containing 90% gets and 10% puts and the other containing 50% gets and 50% puts.

Throughput Comparison: Figure 6.9 shows the achieved operation throughput using a single CPU core for various value sizes in a workload with 90% gets. We can see that the performance of Pilaf far

exceeds that of Redis and Memcached running on top of IPoIB. For small operations (64-byte values) Pilaf achieves ∼1.3 million ops/sec compared to less than 60 Kops/sec for Memcached and Redis. Both Memcached and Redis are bottlenecked by the single CPU core and are unable to saturate the Infiniband card's performance. Because of the CPU bottleneck, their single core performance is the same when running on 1 Gbps Ethernet. We elided those numbers from Figure 6.9 for clarity.

The throughputs of Memcached and Redis can be scaled by devoting more CPU cores to each system. For example, both systems can saturate the 1Gbps Ethernet card when running on four CPU cores. We were not able to scale Memcached and Redis' performance on IPoIB using more CPU cores because the IPoIB driver is unable to spread network interrupts across multiple cores. Nevertheless, even if we optimistically assume perfect scaling, Memcached and Redis require 17× CPU cores to match the performance of Pilaf running on a single core for small key-values. In reality, these systems do not exhibit perfect scaling. For example, [48] reported a 11× throughput improvements for non-batched Memcached puts when scaling from 1 core to 16 cores.

When comparing against Pilaf-VO, we see that Pilaf also achieves substantially better throughput across all operation sizes. In particular, the throughput of Pilaf is 2.1× that of Pilaf-VO for 64-byte values and this performance advantage decreases to 1.1× for 4096-byte values. The shrinking performance gap between Pilaf and Pilaf-VO for larger values reflects the increasingly dominant network transmission overhead for large messages.

Figure 6.9b shows the peak throughput of different systems in a second workload with 50% gets and 50% puts. Not surprisingly, the performance of Memcached and Redis are similar under both workloads.

We were surprised to see that Pilaf achieves identical and sometimes better throughput in the second workload compared to the first. Since RDMA-based get operations have much higher performance than Verb message-based put (Figure 6.1), we initially expected the second workload to achieve worse throughput since it contains a larger fraction of puts. On further investigation, we found that our Infiniband cards appear to be able to process Verb messages and RDMA operations somewhat independently. Quantitatively, the card can reach ∼80% of its peak RDMA throughput while *simultaneously* sending and receiving Verb messages at ∼95% of the peak verb throughput. This explains why the second workload has better throughput. For example, with 256-byte values, the first workload achieves 0.9 million gets/sec (80% of peak RDMA performance) and 0.1 million puts/sec (far less than the card's Verb message sending capacity). By contrast, the second workload produces 0.65 million ops/sec for both get and put which

represents 60% of the card's peak RDMA performance and 94% of the card's Verb message performance. Thus, the second workload has a total throughput of 1.3 million ops/sec, better than that achieved by the first workload.

Latency: Figure 6.10 shows the cumulative probability distribution of operation latencies under different systems in the workload with 90% gets. The underlying experiments involved 10 concurrent clients issuing operations with 1024-byte values as fast as possible.

In Figure 6.10, Pilaf's median latency 15μs, which is determined by the get operation latency. From the earlier experiments in Chapter 2 (Figure 2.3), the average RDMA round trip latency was determined to be ~4μs for 1024-byte reads with a single client. With an average of 1.45 probes (each involving two RDMA reads) to find a particular key-value in a 65%-filled 3-way Cuckoo hash table, the ideal get latency would be 11.2 μs. The extra 4μs reflects the overhead in calculating CRCs on the clients' side plus the queuing effects incurred by having ten connected clients. The latency tail in Figure 6.10 is very short.

As expected, IPoIB also maintains lower latency than Ethernet for both Memcached and Redis. Median Ethernet latency is 209μs for Redis and 230μs for Memcached. Pilaf beats Redis' and Memcached's median Ethernet latency by 14×-15×, and their median IPoIB latency by 9×-11×. The experiments for Figure 6.10 involve ten clients connected to a single server. In these experiments, Pilaf-VO reaches 95% of its peak throughput, Memcached is at 75% of its maximum throughput, and Redis and Pilaf at half their peak throughput. Therefore, queuing effects are uneven for these systems in Figure 6.10. When tested under very light loads (e.g. a single client), Pilaf-VO and Pilaf have similar latency while Memcached and Redis running on IPoIB also have similar latency.

6.6.2 Cell: Other Operations

Like most sorted stores, Cell can perform search, get, put, delete, and range query operations. We explore the throughput of mixes of get and put operations to ensure that Cell retains its advantages for operations more complex than search. While search traverses to a leaf and determines if a key is present in the Cell store:

- get also fetches the value. For server-side search, small values (<1KB) are inlined into the final traversal response. For larger values, a separate Verb message is necessary. With client-side search,

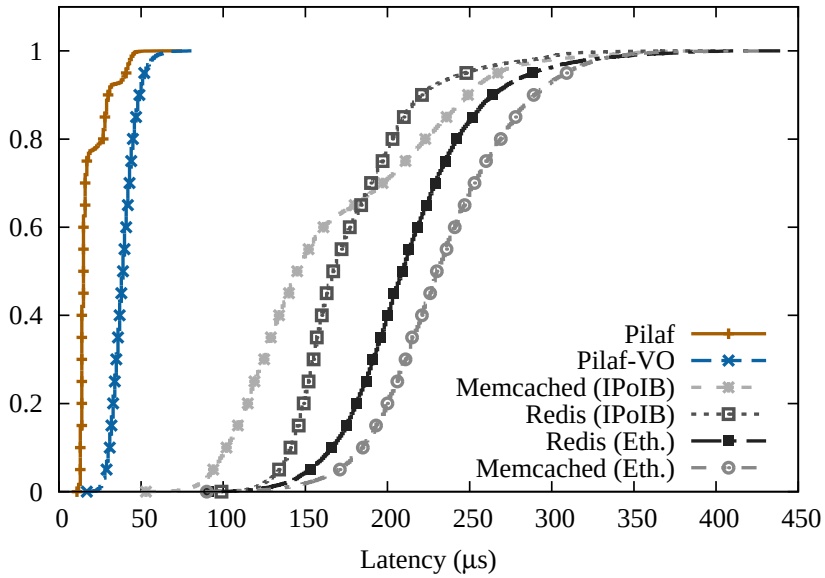


Figure 6.10: CDF of Pilaf latency compared with Memcached, Redis and Pilaf-VO in a workload consisting of 90% gets and 10% puts. The average value size is 1024 bytes. The experiments involved 10 clients.

an additional RDMA read of the extents area is used to fetch the value. This read also fetches a full copy of the key, useful for a potential optimization where key hashes or partial keys are stored in the leaves.

- put performs an additional Verb message exchange to instruct the server to insert the key-value pair into the correct meganode.

Figure 6.11 shows workloads consisting of 0%, 10%, 50%, and 100% get operations and the remainder put operations. Throughput for server-side search only and Cell's hybrid scheme is presented. The increasing Verb message load required for a large ratio of put operations makes the hybrid approach decreasingly effective. In a 0% get (100% put) workload, every operation requires a Verb message for the final insertion, even if the tree traversal was performed with a client-side search. Therefore, a dependency is introduced between the completion of server-side operations and RDMA operations, hence the roughly equal performance of the Cell approach and server-side only search.

Since client-side get operations require an additional RDMA read, we expect Cell's hybrid get throughput to be slightly lower than its hybrid search throughput. Comparing Table 6.1, 1536K get ops/sec can be completed on 4 servers utilizing 2 cores each, while 1626K search ops/sec can be satisfied in the same setup, a loss of 5% throughput.

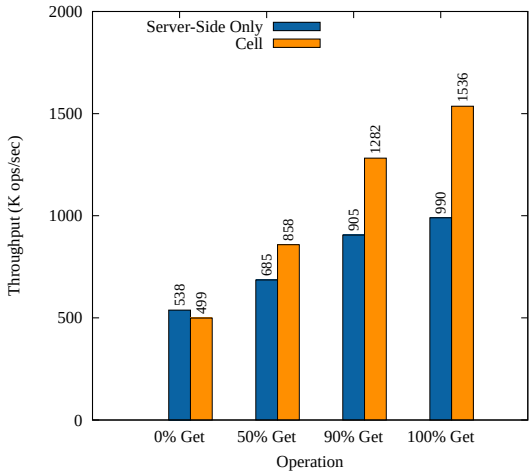


Figure 6.11: Throughput of pipelined mixed get and put workloads on 4 Cell servers, 2 CPUs per server.

6.7 Durable Logging

Both Cell and Pilaf provide crash recovery by logging write operations to durable storage. In the event of a machine failure, the state can be rebuilt by playing back logged writes. We also designed but did not implement techniques for replicated logging using established distributed durable storage systems.

Pilaf: We find that with logging enabled and an average combined key-value length of 39 bytes, Pilaf has no measurable reduction in peak throughput. When the average value length is raised to 1024 bytes, we see a limit of 280K operations per second, precisely matching the 280MBps write capacity of our disk: Pilaf therefore becomes disk-bound. Our servers each have a single SSD, but as suggested in Section 5.1, we can multiplex logging to multiple disks to raise this limit. For our throughput and latency tests, we disable logging for Pilaf and for Redis (Memcached does not include a logging facility). We therefore ensure that any measured bottlenecks are caused by networking or the CPU rather than disk.

Cell: Similarly, Cell becomes bottlenecked on the network throughput or CPU capacity of its servers before the 280MBps append write speed of our SSDs. On our local cluster, Cell can complete $\sim 500K$ writes per second using 2 cores on each of 4 machines. With keys uniformly distributed from 8 to 64 bytes, and values uniformly distributed from 8 to 256 bytes, this yields an average key-value size of 168 bytes (177 bytes with metadata). We therefore require 84.4MBps of SSD sequential write capacity per machine with two cores used, or 42.2MBps per core per machine. For an SSD with a maximum append write speed of 280MBps, Cell could log simultaneously from 6 cores per machine before requiring a second SSD. Experimentally, we find that toggling durable logging produces no measurable difference in throughput; as the Nome cluster lacks SSDs, we leave logging disabled in all of our experiments.

6.8 Summary

The designs for systems exploiting relaxed data-computation locality are well-suited to future datacenter interconnects. The evaluation in this chapter shows that they are also fast, scalable, and practical. Pilaf and Cell both achieve hundreds of thousands to millions of operations per second on a single server or even a single server CPU core with latencies in the tens of microseconds. Each can scale to large clusters of machines. In addition, the self-verifying data structures necessary to detect memory races in the face of relaxed locality provide guaranteed correctness, and the locality selector necessary for selective locality relaxation yields optimal throughput and latency.

7
RELATED WORK

While Pilaf and Cell are among the first distributed systems to fully exploit RDMA and ultra-low latency networking, there has been much work in the HPC and systems communities to exploit performance-critical features like kernel and CPU bypassing. Many MPI implementations, e.g. MPICH/MVAPICH [41, 43] and OpenMPI [66], support an Infiniband network layer, leveraging both Verb messages and RDMA to reduce latency and increase bandwidth. However, the advantages of RDMA and ultra-low latency networking have been explored less thoroughly in distributed systems design.

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, thereby increasing the overall performance in scenarios when the system is bottlenecked by its message-passing capability. We refer to this approach as *RDMA-optimized communication*. The other approach is to explore the CPU-bypassing functionality of RDMAs to improve the performance of systems that are bottlenecked by servers' CPUs. We refer to this approach as *systems with client-side processing*; Pilaf and Cell fall into this category. This section discusses related projects exploring both approaches and reviews work on distributed storage. Systems exploring general RDMA-optimized approaches are presented first, followed by work specifically relevant to Pilaf and Cell individually.

7.1 RDMA-Optimized Communication

RDMA-optimized communication that still uses the CPUs on both ends for each message has been the subject of a decade of research. The HPC community has explored [70] and exploited the performance advantages of RDMA extensively to improve the MPI communication substrate. There are several RDMA-optimized MPI [44, 41, 43, 42, 67] and OpenMPI [66] implementations that are widely used. Recently, the systems community has begun to explore the use of RDMA in distributed systems. Most projects focus on using RDMAs to improve the underlying message-passing substrate of systems such as in-memory key-value caches [32, 31, 30, 69], HBase [25], Hadoop [45], PVFS [76] and NFS [19]. All of these works port existing system designs to a modified networking backend that utilizes RDMA within the traditional request/reply message exchange pattern. In other words, RDMA is used as a supplemental mechanism

to optimize data transfers while verb or other messaging mechanisms are required before each RDMA to signal control information before the transfer. HERD has proposed using an RDMA write to send a request to the server and to have the server respond using an unreliable datagram (UD) message [32]. FaRM also uses RDMA writes to implement a fast message-passing primitive [16]. By contrast, our work aims to replace a large fraction of the request/reply message exchanges with one-sided RDMA reads by the clients, thereby significantly reducing the server's CPU overhead.

The advantage of using RDMA to accelerate message passing is that the resulting solutions are generally applicable, because all distributed systems could use such a high performance message-passing primitive for communication. However, in scenarios where the server's CPU becomes the bottleneck, instead of the underlying network I/O, this approach does not take full advantage of the CPU-bypassing capability of RDMA to relieve servers' CPU load.

Due to the perceived cost of specialized HPC hardware, some in the systems community have advocated software RDMA over traditional Ethernet. Soft-iWARP is a version of the iWARP protocol implemented entirely in software [71]; it reduces TCP latency by 5%-15% by minimizing data copying and limiting the number of context switches required. Another project later used soft-iWARP to realize a 20% reduction in per-get CPU load for Memcached without Infiniband hardware [69].

7.2 Pilaf

As they represent a vital component in many datacenter-scale systems, key-value stores are the subject of much current research effort. Most key-value stores map each flat key to a flat chunk of value data, and contain either text or numeric data. In the era of so-called "Big Data", data frequently exceeds the capacity of a single machine, so key-value stores are usually distributed. For example, Amazon's Dynamo system is a distributed key-value store that offers high reliability in exchange for eventual consistency [15]. Many distributed computing frameworks implement some form of a distributed key-value store. The widely used MapReduce framework processes data as sets of key-value pairs [14], while the Piccolo and Oolong distributed computing frameworks implement an explicit key-value store with user-specified data-computation locality [59, 53]. Silt combines an in-memory index with extents stored on SSDs for a key-value store as fast as and larger than RAM [39]. MICA uses a plethora of techniques including replacing TCP connections with UDP, using Intel's DPDK [27] to move packet processing to

CPUs, and careful awareness of NUMA costs [40]. The system achieves 8.2M-9.6M small key-value ops/sec per 10Gbps Ethernet NIC and 2 server cores on 8-NIC, 16-core machines.

Key-value stores are an obvious target for RDMA-enabled infrastructure. Three projects use RDMA writes to accelerate standard message-passing designs and implement a version of Memcached [31, 30, 69]. A fourth system, FaRM [16], uses RDMA writes to implement a fast message-passing primitive that entirely replaces Infiniband's own message-passing capabilities. A fifth system, Nessie [7], takes the design path that we avoided in also using RDMA write and atomic operations to implement client-driven write operations.

The three projects that implement Memcached over RDMA on Infiniband [31, 30] or Soft-iWARP [69] also adopt the usual combination of control messages plus RDMAs write to process gets and puts at the server. In [31], the client uses a Verb message to send a local buffer address to the server, into which the server then copies data using an RDMA write. put operations also require two Verb messages and one RDMA read: the client gives the server an address, from which the server pulls a key-value pair via an RDMA read. Both put and get include short-operation optimizations that combine the data normally read or written via RDMA into one of the Verb messages exchanged. Compared to Pilaf, this design achieves much lower throughput. Their reported performance in a Infiniband cluster similar to ours is 300 Kops/sec for small operations, significantly lower than that achieved by Pilaf (1.3 million ops/sec). The other Memcached over Infiniband project [30] combines Infiniband's Reliable Connection (RC, with guarantees similar to TCP) and Unreliable Datagram (UD, resembling UDP) modes. The resulting performance is also lower than achieved by Pilaf, despite it running on a QDR Infiniband cluster that is twice as fast as our DDR Infiniband interconnect.

FaRM [16], roughly contemporaneous with Pilaf, offers a generalized solution for in-memory distributed storage systems exploiting RDMA for extremely high throughput. It uses RDMA reads for lock-free lookups, and RDMA writes to provide high-speed message passing. One of the sample applications built on FaRM is a key-value store, optimized to satisfy hash table lookups in a single RDMA read whenever possible. By bounding key-value lengths and inlining key-value pairs into the hash table, FaRM can satisfy key-value lookups in an average of 1.04 RDMA reads at 90% occupancy. FaRM's "chained associative hopscotch hashing" algorithm provides faster reads at higher occupancy than Pilaf, although for key-value pairs of highly variable or unbounded size that cannot be inlined, Pilaf's approach requires only 1.12 additional RDMA reads per key-value pair [16].

Nessie [7], published a year after Pilaf was presented, goes one step further. Instead of offering client-side reads and server-side writes, Nessie pushes all operations to the client. Pilaf uses N=3 cuckoo hashing, while Nessie uses N=2 cuckoo hashing, where each key has one primary and one alternate slot. Read retries are thus minimized, at the cost of more frequent collisions when inserting new key-value pairs. If the primary and secondary slots are filled with other keys, and the alternate slots for those keys are filled, then Nessie resorts to a server-driven table restructuring or resize. Like Pilaf, Nessie disconnects clients to prevent erroneous memory accesses while server-driven table resizes are underway. In addition, Pilaf's server-side writes make it easy to perform chained migration without the need to resize the table or disconnect clients. Finally, Pilaf cannot leave the hash table in an inconsistent state if a client crashes during a multi-stage write operation. Although Nessie is a write-optimized key-value store, the average number of RDMA round trips required to complete a write operation increases from 2.5 at a load factor of 10% to 9.9 at 75%. Since RDMA operations are fast, the average write latency at 75% occupancy is still $< 100\mu s$, compared to Pilaf's $60\mu s$ latency for 40 clients sending put operations to a single-core Pilaf server (and $15\mu s$ for 10 clients). Although not explicitly mentioned in the system description, Nessie's design diagrams indicate that the key-value extents are fixed-length, while Pilaf allows for key-value pairs of arbitrary size (as shown in Figure 3.2).

7.3 Cell

Distributed sorted stores are another important substrate for datacenter-scale applications. Cell is a fast, CPU-efficient sorted store for HPC-like networks. Cell is inspired by FaRM and Pilaf to offload the processing of read-only requests to the clients via the use of RDMA reads. In the context of B-tree traversal, client-side searches become desirable when server CPUs become bottlenecked and adds unnecessary latency otherwise (due to multiple rounds of RDMAs). Cell allows clients to dynamically decide whether to use client-side or server-side processing depending on measured server performance.

7.3.1 Distributed B-trees

There are two high-level approaches for constructing a distributed B-tree. One builds a distributed tree out of lean nodes, e.g. 8KB nodes in Boxwood [47] or 4KB nodes in [3] and Minuet [68]. Small nodes are crucial in the design of these systems because they 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 [8], builds a tree out of fat nodes (up to 200MB) [23]. This design reduces the number of different machines that each search needs to contact but requires server-side processing to search within a fat node. Cell takes a hybrid approach: it builds a global tree of fat nodes in order to minimize search latency when server CPUs are plentiful and builds a local tree of lean nodes to allow RDMA-based client-side processing when server CPUs become bottlenecked.

Apart from the hierarchical design, Cell also differs from prior systems in how it handles concurrency. Both Johnson and Colbrook [28] and Boxwood [47] 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 caches internal nodes consistently across servers, resulting in a complex scheme that requires distributed locks. By comparison, Cell's implementation of Sagiv's tree is simpler and 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. [3] implement a regular B-tree and handle concurrency requirements using distributed transactions. These transactions are more conservative than necessary and are especially heavy-handed for read-only searches. Minuet [68] is based on Aguilera et al.'s system. It addresses some of the scalability bottlenecks and additionally provides multiversioning and consistent snapshots. Some of their improvements emulate the B-link tree, such as their use of fence keys (similar to Cell's per-node minimum and maximum keys) yet they do not seem to benefit from the simplicity of Sagiv's single-locking scheme.

7.3.2 In-Memory Distributed Storage

The high latency associated with disk-based storage has led to a research effort behind in-memory storage systems. H-Store [33, 29] and VoltDB [74] are distributed in-memory databases. Instead of a distributed B-tree, both use a simple hash-based partition scheme to spread the underlying data across the memory of many machines. Masstree [48] and Silo [72] provides fast multi-core single server in-memory B-tree implementations; they are not based on B-link trees.

8
CONCLUSION

Datacenter networks are poised to adopt feature previously found only in high-performance computing interconnects. These new network features necessitate rethinking the designs of fundamental distributed systems, especially distributed stores like key-value stores and sorted stores. To take full advantage of low-latency round trips and RDMA, we have designed prototype distributed storage systems around two key insights about new datacenter interconnects: (1) relaxing locality between data and computation is practical; (2) *selectively* relaxing locality improves load balancing.

This thesis demonstrates Insight 1 with a distributed in-memory key-value store with moderate server CPU overhead. Pilaf moves computation for all read operations from the server to the client by having clients directly read from the server's memory using RDMA. As justified in Chapter 1, repeatedly retrying conflicting write operations carries a higher computation cost than retrying read operations. Therefore, Pilaf's design follows conventional wisdom in using server-side writes to send write computation to the data. Pilaf uses *self-verifying* data structures to detect read-write races in the face of concurrent RDMA reads performed by the clients and the local memory accesses at the server. Pilaf is able to achieve a peak throughput of over 1.35M read and 663K write operations per second from a single CPU core, outperforming existing systems running over Ethernet or IPoIB by more than an order of magnitude. We refined our approach with Cell, a distributed in-memory sorted B-tree store. Cell follows Insight 2, selectively decoupling data and computation locality. It keeps computation and data together at the servers under low load, and moves read computation to the client when servers are under high load. Cell effectively balances the use of client-side and server-side search. It achieves high throughput at low latency with primarily server-side search under low server load, and maintains liveness and low latency when server CPUs are bottlenecked by moving computation to clients' CPUs.

The lessons learned in designing and building Pilaf and Cell lead to the concrete contributions this thesis adds to the state of the art:

- **CPU-efficient systems relax data-computation locality:** Moving some or all computation from the server to the client allows systems to utilize a combination of the available client and server CPU resources.

- **Self-verifying data structures make RDMA-capable systems practical:** Relaxed data-computation locality via RDMA introduces the problem of memory access races between RDMA reads and server CPU writes. Self-verifying data structures allow clients to correctly detect conflicts.

- **Selectively relaxing locality yields better load balancing:** Utilizing server CPUs when load is low allows systems to extract the low latency associated with traditional data-computation locality. When load peaks, systems maintain low operation latency by shifting computation to clients, using RDMA to bypass servers' CPUs.

- **Locality-relaxation techniques work at scale:** The prototype systems in this thesis achieve high throughput and ultra-low latency at scale by relaxing locality on all or some read operations. Pilaf, a distributed key-value store, achieves double the throughput of a server-side search system by offloading all read operations to clients' CPUs. Cell, a distributed sorted B-tree store, consistently achieves 75% to 90% higher performance than server-side only searches on 2 cores per server on clusters of 1 to 16 server machines.

We believe that future distributed systems built for next-generation datacenter networks will incorporate these insights and observations, selectively relaxing data-computation locality using RDMA to achieve optimal performance and superior load-balancing. In addition, we envision that these contributions will inform the architecture and features of future datacenter interconnects. Infiniband serves as a model of the datacenter network of tomorrow in this thesis, but it remains to be seen what fabric will be used in future datacenters. It is likely that ultra-low latency ($< 3\mu s$) via RDMA and kernel-bypassing and high bandwidth (≥ 40 Gbps) will by available, and the relaxed-locality techniques embodied in Pilaf and Cell could be effectively used with any such interconnect. Applications in tomorrow's datacenter will require more, faster in-memory storage, and systems that relax data-computation locality to efficiently utilize CPU resources and balance load fit these needs.

BIBLIOGRAPHY

[1] Extending high performance capabilities for microsoft azure, 2014.

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3):5:1–5:48, November 2009.

[3] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Proc. VLDB Endow.*, 1(1):598–609, 2008.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Kevin J Barker, Kei Davis, Adolfy Hoisie, Darren J Kerbyson, Mike Lang, Scott Pakin, and Jose C Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 1. IEEE Press, 2008.

[6] Philip Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Survey*, 13(2):185–221, June 1981.

[7] Ben Cassell, Tyler Szepesi, Bernard Wong, and Tim Brech. Designing a low-latency cuckoo hash table for write-intensive workloads using rdma. In *Proceedings of First International Workshop on Rack-scale*, 2014.

[8] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

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

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012.

[12] CD Cuong. Youtube scalability. In *Google Seattle Conference on Scalability*, 2007.

[13] Jeff Dean. Software engineering advice from building large-scale distributed systems. Slides.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[17] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[18] S. Ghemawat and J. Dean. Leveldb, 2011.

[19] Garth Gibson and Wittawat Tantisiriroj. Network file system (nfs) in high performance networks. Technical report, Carnegie Mellon University, 2008.

[20] J. Gray and A. Reuter, editors. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[21] Frank Haight. Two queues in parallel. *Biometrika*, 45(3), 1958.

[22] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

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

[24] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[25] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi ur Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High-performance design of HBase with

rdma over infiniband. 2012.

[26] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[27] Intel. Intel data plane development kit (inteldpdk), 2014.

[28] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. Technical Report MIT/LCS/TR-530, Massachusetts Institute of Technology, 1992.

[29] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, 2010.

[30] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. 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.

[31] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, 2011.

[32] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306. ACM, 2014.

[33] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[34] Darren J Kerbyson. A look at application performance sensitivity to the bandwidth and latency of infiniband networks. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7–pp. IEEE, 2006.

[35] Mike Krieger. What powers instagram: Hundreds of instances, dozens of technologies.

[36] R. Kutzelnigg and M. Drmota. *Random bipartite graphs and their application to Cuckoo Hashing*. PhD thesis, PhD thesis, Vienna University of Technology, 2008.

[37] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system.

SIGOPS Oper. Syst. Rev., 44(2):35–40, April 2010.

[38] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[39] H. Lim, B. Fan, D.G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.

[40] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[41] J. Liu, W. Jiang, P. Wyckoff, D.K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16, april 2004.

[42] J. Liu, W. Jiang, P. Wyckoff, D.K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16, april 2004.

[43] Jiuxing Liu, Jiesheng Wu, Sushmitha Kini, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit Noronha, Pete Wyckoff, and Dhabaleswar Panda. Mpi over infiniband: Early experiences. In *Ohio State University Technical Report*, 2003.

[44] Jiuxing Liu, Jiesheng Wu, Sushmitha Kini, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit Noronha, Pete Wyckoff, and Dhabaleswar Panda. Mpi over infiniband: Early experiences. In *Ohio State University Technical Report*, 2003.

[45] Xiaoyi Lu, N.S. Islam, M. Wasi-ur Rahman, J. Jose, H. Subramoni, Hao Wang, and D.K. Panda. High-performance design of hadoop rpc with rdma over infiniband. In *International Conference on Parallel Processing (ICPP)*, 2013.

[46] Patrick MacArthur and Robert D Russell. A performance study to guide rdma programming decisions. In *2012 IEEE 14th International Conference on High Performance Computing and Communication (HPCC),*, pages 778–785. IEEE, 2012.

[47] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *USENIX Symposium on Operating System Design and Implementation*, 2004.

[48] Y. Mao, E. Kohler, and R.T. Morris. Cache craftiness for fast multicore key-value storage. In

Proceedings of the 7th ACM european conference on Computer Systems, pages 183–196, 2012.

[49] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, 2015.

[50] Mellanox. Rdma aware networks programming user manual.

[51] Mellanox. Infiniband credit-based link-layer flow-control, 3 2014.

[52] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in CryptologyCRYPTO87*, pages 369–378. Springer, 1988.

[53] C. Mitchell, R. Power, and J. Li. Oolong: asynchronous distributed applications made easy. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 11, 2012.

[54] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, 2013.

[55] Rajesh Nishtala, Hans Fugal, Steve Grimm, Marc Kwiatkowski, Herman Lee, Harry Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkat Venkataramani. Scaling memcached at facebook. In *Proceedings of USENIX NSDI 2013*, 2013.

[56] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *USENIX Annual, FREENIX Track*, pages 183–191, 1999.

[57] WHM Oortwijn. Distributed symbolic reachability analysis. 2015.

[58] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[59] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[60] M.J. Rashti and A. Afsahi. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[61] Yehoshua Sagiv. Concurrent operations on b*-trees with overtaking. *J. Comput. Syst. Sci.*, 33(2):275–296, 1986.

[62] S. Sanfilippo and P. Noordhuis. Redis, 2011.

[63] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 2011.

[64] Siddhartha Sen and Robert E. Tarjan. Deletion without rebalancing in multiway search trees. *ACM Trans. Database Syst.*, 39(1):8:1–8:14, 2014.

[65] Dennis Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90, March 1988.

[66] G.M. Shipman, T.S. Woodall, R.L. Graham, A.B. Maccabe, and P.G. Bridges. Infiniband scalability in open mpi. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp, 2006.

[67] G.M. Shipman, T.S. Woodall, R.L. Graham, A.B. Maccabe, and P.G. Bridges. Infiniband scalability in open mpi. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp, 2006.

[68] Benjamin Sowell, Wojciech M. Golab, and Mehul A. Shah. Minuet: A scalable distributed multiversion b-tree. *Proc. VLDB Endow.*, 5(9):884–895, 2012.

[69] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of USENIX Annual Technical Conference*, 2012.

[70] Sayantan Sur, Matthew J Koop, Lei Chai, and Dhabaleswar K Panda. Performance analysis and evaluation of mellanox connectx infiniband architecture with multi-core platforms. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 125–134. IEEE, 2007.

[71] A. Trivedi, B. Metzler, and P. Stuedi. A case for rdma in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 17, 2011.

[72] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[73] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N.S. Islam, H. Subramoni, and D.K. Panda. 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*, pages 48 –55, aug. 2012.

[74] LLC VoltDB. Voltdb technical overview. odbms.org/download/VoltDBTechnicalOverview.pdf, 2010.

[75] Wayne Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14(1), 1977.

[76] J. Wu, P. Wyckoff, and D. Panda. Pvfs over infiniband: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 125–132, 2003.