BlueBridge: Pushing Shared Memory Management into the Network

Fabian Ruffy
University of British Columbia
fruffy@cs.ubc.ca

Amanda Carbonari
University of British Columbia
acarb95@cs.ubc.ca

ABSTRACT

Distributed shared memory (DSM) systems have been well explored, but never managed to reach widespread adoption due to the complexity involved in building them. Instead of attempting to use or build a complex DSM system, programmers opted to control the messaging semantics of their applications. An increasing trend of in-memory computation and change in datacenter programming models has led to a resurgence of DSM. Disaggregated racks require the application to reason about shared memory in order to leverage the large amount of memory in a dedicated server. DSM systems can improve the programmability of these applications, but all the management complexity inherent to DSM systems must still be addressed.

To mitigate this complexity, we propose BlueBridge, a network managed shared memory system which stores every memory address as an IPv6-compliant address. This allows the location information of memory to be encoded in the IPv6 header, which the switch can operate on, effectively moving the directory service to the switch. With the global knowledge of where memory resides and the traffic information, the controller can now manage access, migration, and concurrency of the memory.

We realize BlueBridge with an initial proof of concept implementation, which shows that IPv6 memory addressing is possible. When evaluating this implementation, we show that we can achieve a 2x reduction in latency per remote page access and, with RDMA, we can achieve latencies similar to local disk access. These numbers are not concrete representations of the speed up BlueBridge can achieve, rather they show the promise of what a fully optimized system can achieve.

KEYWORDS

Distributed Shared Memory, Software-Defined Networks, Remote Memory

1 INTRODUCTION

Distributed shared memory (DSM) is already a well explored field [1, 4, 6, 17, 19, 21], showing benefits of scale, improved programmability, and cost effectiveness. DSM systems attempt to reduce complexity of distributed or parallel programming by providing a single address space for multiple processes to use. It handles all communication semantics and management of the memory. This scales well as more memory/nodes can be added easily depending on the system design. The improved programmability of the system comes from the single virtual shared memory address space it provides and the shielding of programmers from often complex communication and management implementations. DSM systems are also cost effective as, at the time, multiprocessor hardware or hardware which could create a shared address space was expensive. [21]

But DSM never managed to reach widespread adoption due to complexity of tracking memory locations, overheads of communication, and complexity of implementing concurrency management [12]. The implementation of application specific messaging proved to be a better trade-off than using a general distributed shared memory system. Writing application specific messaging allows the programmer more control over the communication protocol, data movement, and performance of the system.

So why even bother discussing DSM systems? DSM systems need to be revisited because the traditional programming model which rendered DSM systems impractical is changing. Current datacenter programming models are shifting from traditional servers to rack-scale disaggregated servers. Disaggregated datacenters partition racks into compute units and storage units. The compute unit can be a server with multiple CPUs or GPUs and a small amount of local memory. The storage units are servers with a small CPU and a large array of SSDs, disks, or RAM. [11]

The change in programming model allows for applications to easily leverage large amounts of remote memory. Unfortunately, most applications are written for servers which comprise of some memory and a reasonably adequate CPU. Disaggregation causes the memory units to have small CPUs and the CPU units to have small amounts of memory, something typical applications are not designed for. In-memory computation, latency sensitive systems, and real-time analysis systems all require large amounts of memory and fast compute. These systems could become easier to program and design if they could leverage the disaggregated memory. Considering the change in programming model brought on by datacenter trends, it is imperative to revisit distributed shared memory in this context. But, the complexities surrounding building DSM systems still exist in this new context.

To mitigate the management complexities inherent to DSM systems, we propose BlueBridge, a network managed memory system. BlueBridge leverages advancements in automatic network configurations with IPv6 addressed memory at the core of its design. By storing every memory address in an IPv6-compliant manner, the location information of remote memory is encoded in the IPv6 packet header, which makes it available to the switch when routing. Software-defined networking (SDN) allows the controller of the switch to have a global view of the network, thus a global view of the memory system. It can determine hot spots, migrate memory, etc. by installing rules on the switch or monitoring the traffic on the switch.

This design eliminates migration complexity, the switch can reroute the address to the new machine, and reduces the amount of management communication. Typical distributed shared memory systems always require a directory service to determine where the memory is stored. Requiring the switch to act as this directory service removes an extra roundtrip time. It does not directly solve the complexity surrounding concurrent access, but, with a global view of the system, the controller can implement policies to make reasoning and solving concurrency less complex.

We realize this design in a proof of concept implementation, which shows that IPv6-compliant addressing is possible and can be used to access remote pages during a page fault. We evaluate this implementation by exploring the breakdown of latency per page fault and comparing the latency to local disk. We find that by moving the directory service into the switch, we can improve the latency by 2x and, by leveraging RDMA communication, we can perform a remote page fault in a similar latency to a local disk page fault. These numbers are not concrete due to the proof of concept nature of our implementation, but they show the promise of integrating the network with the shared memory management.

Our paper is organized as follows. First we discuss the related works for this project in Section 2. Next, we discuss the full design of BlueBridge in Section 3. Section 4 details our proof of concept implementation, and Section 5 evaluates that proof of concept. Finally we conclude with future works in Section 6.

2 RELATED WORK

We categorize related work to BlueBridge in three categories: distributed shared memory systems, remote paging systems, and using Software Defined Networking for storage or memory.

Distributed Shared Memory. Distributed shared memory has been explored extensively in previous decades [1, 4, 6, 17, 19, 21]. Our solution is similar to traditional distributed shared memory implementations but differs fundamentally in the management strategy. Therefore, we restrict our discussion of related works to the management components of distributed shared memory systems.

Two of the most recent systems are FaRM from Microsoft[4] and Grappa from University of Washington [17]. Both are similar to the traditional approaches described in [19, 21] but differ in the hardware used and design decisions. Both Grappa and FaRM take a more decentralized approach to their directory services. Each node is required to contact the "owner" of the memory in order to get routing information or to send requests [4, 17]. This solution tends to avoid the performance overheads from a centralized solution, but can perform poorly when memory migrates. Our solution differs as the memory address is the IPv6 location of the memory. The switch will map that IPv6 address to the correct machine which owns that piece of memory. In the case of memory migration, the controller will update the switch with new rules, thus re-routing the same IPv6 address to the new owner.

Remote Paging. Remote paging works almost identically to typical operating system paging, except instead of going to disk, it fetches the data from a remote memory source. A lot of work has been done exploring this area, generally, it is found that remote paging tends to be faster than disk accesses when a workload's data cannot fit entirely on its local RAM [7]. Two remote paging solutions similar to ours leverage the network [7, 14] to improve performance. Infiniswap uses RDMA and Infiniband and achieves better performance on a set of unmodified applications using decentralized placement and eviction. Our solution differs from Infiniswap in two ways: 1) the memory we handle is *shared* as well as remote, and 2) we not only leverage the network, but integrate with it. Infiniswap leverages the network for their implementation, but does not ascribe any intelligence to it, we actually attempt to move intelligence into the network for performance gain and complexity reduction.

Software-Defined Networking for Storage/Memory. In the past years, several systems have utilised SDN to shift traditional host-based management into the network. SwitchKV [13] uses programmability to implement cache look-up directly on the the switch. The controller places table entries containing information whether data is present in the cache node. If not, requests will be routed accordingly and subsequently cached on the node as well as the switch table. Our use of software-defined networking is similar to SwitchKV as we leverage the global knowledge benefits of the control plane. However, we do not only re-route packets, but incorporate any form of management directly in the networking layer. This may also include validation, fault tolerance, and coherence.

3 BLUEBRIDGE DESIGN

In the following section, we outline our design approach of the BlueBridge shared memory system. This roadmap is by no means final and will be subject to change and amendments during realization. Scope as well as generalizability of the system are not yet determined, and may vary to accommodate particular application types. Our assumptions about network and eventual switch configuration capabilities are drawn from the OpenFlow Switch specification v1.50 [20] and the P4 programmable switch paradigm [3].

3.1 Assumptions

As BlueBridge is intended to run in constrained and controlled datacenter environments we make several assumptions about the environment and underlying technology of the project.

Communication in a distributed shared memory system should be inexpensive and performant. Ideally, a remote operation provides a substantial advantage over disk-reads and the penalty compared to a local memory access is minimal. As a consequence, we expect a working prototype to be able to support RDMA over Converged Ethernet (RoCEv2) as transmission protocol for our remote paging requests to be satisfactory. We are confident in these assumptions, as recent research has demonstrated the potential of this comparably modern technology. [8]

Second, we require our system to run in a reliable network environment with state-of-the-art network components. We assume that link-speed as well as switch capabilities are sufficient to handle the communication load of the distributed system. We do not take packet loss into consideration and expect reliable transmission from each node.

Third, we currently do not intend to scale beyond a 42-U rack size managed by a single ToR switch. Every rack is an autonomous cluster computationally independent of other entities in the network. Furthermore, any packet exchange between nodes in the cluster will only requires a single hop over the managing switch. For our current approach we also disregard extensive replication

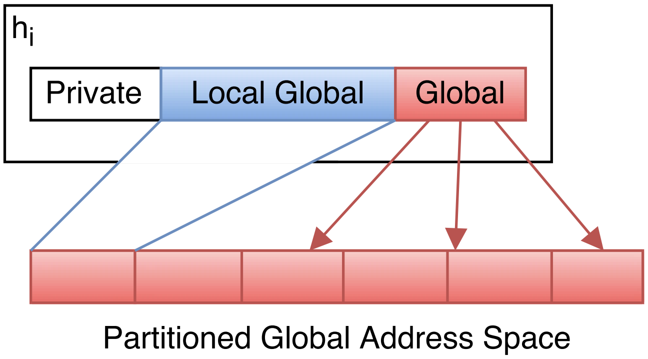

Figure 1: Illustrations of how global memory is stored in a BlueBridge host.

and high fault-tolerance techniques. As a single point-of-failure, a malfunctioning switch or controller will imply a total system failure.

3.2 Proposed Design Overview

The eventual design goal of BlueBridge is to implement a software-based virtual memory system in conjunction with a distributed memory platform. The approach is founded upon a partitioned global address space (PGAS), 4096 bytes page implementation. Each machine's memory section contains a private and a global address space (see Figure 1). Private memory is only accessible to critical or privileged processes of the local host, it is not shared with any other node. Global address space is divided into a local and remote section, the former residing physically on the machine. Local global memory encompasses the address space a server provides for other machines to use, it may also contain memory the host has fetched from remote machines and stored for computation. The global address space is comprised of a collection of pointers the local machine has either reserved from remote hosts or requested for future operations. In order to preserve locality, hosts are aware of the global memory available on the machine and all operations are performed locally. In case of a remote fetch, data is written back to the original host. In a scenario involving disaggregated memory and servers acting as memory banks, remote memory will treated as a single uniform address space. BlueBridge follows a simple design concept, as shown in Figure 2.

The core of the proposal is a divergence of conventional memory address properties. BlueBridge does not rely on typical PGAS memory design and directly encodes memory addresses as IPv6 pointers. Each client acquires remote memory in the form a 128 bit IPv6-conform address. Instead of storing a request in the payload of a network packet, any subsequent request will be directly inserted into the destination header of the packet. This design choice allows for powerful operations in the network space which may greatly simplify consistency and coherence management in a DSM system. Any request or operation can be routed, forwarded, copied, or modified directly without hosts being required to track state. Both receiver and sender remain fully unaware of the remote location or current state of the memory as these operations are handled by dynamic network functions.

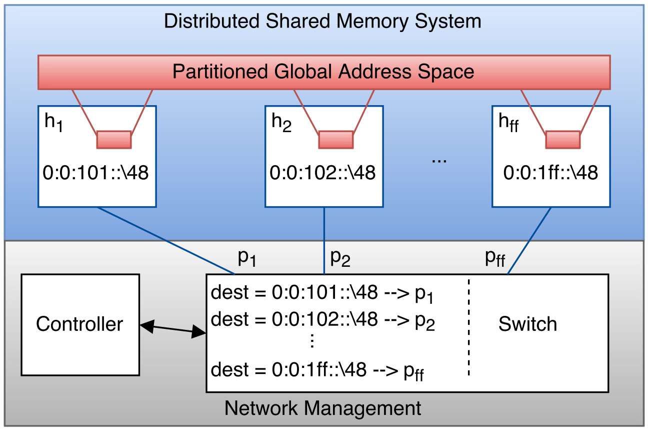
Distributed Shared Memory System
Partitioned Global Address Space
h_1
0:0:101::\48
h_2
0:0:102::\48
...
h_{ff}
0:0:1ff::\48
p_1
p_2
p_{ff}
Controller
dest = 0:0:101::\48 --> p_1
dest = 0:0:102::\48 --> p_2
⋮
dest = 0:0:1ff::\48 --> p_{ff}
Switch
Network Management

Figure 2: High-level blueprint of BlueBridge architecture.

To support the architectural concept we utilize a software-defined switch as well as controller handling all required operations. To avoid overloading the switch, entries are kept minimal. Servers are assigned a subnet range which represents a global memory partition. In combination with the pointer address, this subnet range creates an unique identifier in the packet header.

If a client performs an operation requiring the use of a remote pointer, it will insert the stored pointer into the destination IP field of the IPv6 header and send the packet. Since the header contains the unique node ID, the switch is able to instantaneously forward the packet to the correct port. The server will extract the pointer and specific command out of the destination header, perform the specified operation, and reply to the source IP of the packet (which is the unique ID of the client). A client “keeps track" of an address by extracting the source ID identifier of the replying owner. Neither server nor client are required to store any information about the location of other machines. For n servers the switch only has to maintain n basic routing entries, mapping each subnet to a specific port.

3.3 Operations

The client library currently consists of simple CREATE, READ, UPDATE, DELETE (CRUD) functions for a higher-level application to utilise. The current operational plan for each procedure is as follows:

CREATE. If a client needs to allocate or reserve a remote memory section, it will call CREATE. The function selects a random ID of the available server range and sends a IP packet containing the create command and destination ID. The switch forwards the packet to the corresponding server. After allocation of a local pointer has been successful on the server side, it stores the address in the payload of the packet and responds with an ACK message. The client will extract the pointer as well as the unique identifier of the server, merge the structure into a unique global pointer, and store it for future use.

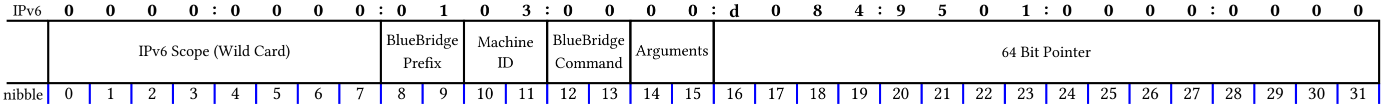

Figure 3: The IP destination structure. The top row represents a sample IPv6 request which is mapped to the structure below.

The size of a single allocate operation is limited to 4096 byte pages. It may be beneficial to allow clients the specification of arbitrary allocation size. If the size exceeds a certain threshold, both parties can switch to a TCP connection for more efficient data transfer.

For future work, we are considering to render clients fully agnostic of the destination and introduce load-balancing capabilities on the switch. Clients will send generic CREATE destination requests, which will be distributed by the software switch based on optimal locality or load. This approach is possible as the switch is able to forward the generic packet to a controller, which could decide the current optimal placement. It remains to be seen if the trade-off between additional request latency and efficient placement will provide long-term gains.

We are also considering the use of broadcast allocation to inform all other clients of a newly available memory address. Information about the location of shared variables can thus be quickly spread through the network.

WRITE. Once a client has acquired a pointer, it is able to write to the remote address by inserting the IPv6 structure into the packet header and coupling it with an WRITE command. Data that is to be written is encapsulated in the payload. The server will extract the pointer, store the data of the payload in its global address space, and acknowledge to the client that the operation has been successful. We are currently intending to support three different types of write commands natively.

- *Allocate Writes.* The client will insert a ALLOCATE-WRITE command in the header as well as the data content in the payload. The server will write the data to its local memory and provide the allocated memory address to the client.
- *Blind Writes.* Blind writes are simple asynchronous operations unaware of any consistency or coherence semantics. The content of a blind-write packet is directly written to the server disregarding any that has been previously stored in the block. Applications may use this type of write at their own risk.
- *Updates.* Updates are transactional procedures which guarantee data consistency and correctness. An update operation requires a READ before being able to write data. Update operations transfer ownership temporarily to the accessing node. Further semantics are described in the READ section.

READ. A READ operation is functionally similar to the WRITE call. The server will extract the pointer, store the memory content in the payload, and send the data to the client.

In a concurrent and asynchronous implementation involving direct memory access it may occur that an other client has fetched the same memory and is currently modifying the data block. Our approach to solve this potential race condition is as follows:

As soon as a client has requested an UPDATE operation on a particular memory address, the switch will install a temporary redirect route forwarding all subsequent requests to the current owner of the memory section. The new owner may either reject or process the requesting, depending on the consistency policy of the system or application. To prevent stale reads before the redirect is updated, the server will invalidate its memory section until the client has rewritten the data. After a successful procedure, all temporary flows are removed and data exchange continuous as usual.

DELETE. A DELETE operation is a simple `free` or `memset(0)` operation on the server side depending on desired outcome. If the memory is to be re-used, it can be set to zero, if it is supposed to be deallocated, it can be freed. The client can call DELETE at any time. To prevent the remaining machines from using a stale pointer we are planning to utilise a broadcast message containing the memory address in question. The switch forwards the message to all machines in the network and any affected client will invalidate its pointer.

All of these operations are are stored in the IPv6 header of the packet encoded as single byte, providing a total of 127 unique commands. Storing commands in the header also allows the central coordinating switch to reroute or reject packets based on the nature of the request without involving a server. This feature also opens up the possibility of the control plane to monitor the computational state of the system and interject if necessary.

3.4 Control Plane

As the majority of management and coordination operation are shifted into the network, the controller as well as the software switch have to be able to bear a significant amount of computation and load. The amount of network nodes a packet traverses until its destination has to be minimal. Avoiding a flow table "cache miss" is desired as much as possible is desired as flow-table updates are expensive. These aspects require a proactive flow-installation system and the controller to have global knowledge of the memory layout. Flow table entries in the switch need to cover most common events. In the default case, the controller will monitor active nodes in the system and proactively install corresponding flow entries. Any packet that matches a particular IPv6 CIDR address will be forwarded to the owner without any further overhead.

Update operations. Update operations require consistency which can be enforced by the control plane. For any header containing an UPDATE command, the switch will forward the packet to the node as well as the controller to process. The packet will trigger the controller to call a flow-table update procedure which will install a redirect entry of the specific pointer address to the new owner. In order to mitigate the exhaustion of flow tables, these entries must be temporary. After installing the flow rule, two possible events may occur. Either the client writes the content back to the original node and triggers a new controller action to delete the entry, or the flow expires. If a flow expires, the controller as well as server assume that the client operation has failed and the memory section is marked available again. The delayed write operation of the client will be rejected.

Monitoring. Port mirroring of switch traffic as well as out-of-band monitoring and management of client machines allow for a comprehensive central view of the controller of all network events. Opportunities may include fast-failover in case of single machine failure, or identification and dynamic rebalancing of hot memory partitions.

In addition to conventional strategies, a realizable migration concept may include live-updating of flow entries to move traffic to less active nodes. As the switch is able to route by inspecting commands in the destination header, it is possible to redirect WRITE operations to a different host. After successful completion, the controller will install a permanent flow entry redirecting requests to the new owner. A client thus incidentally and seamlessly migrates data while keeping the data flow of the system intact. One significant drawback of this method is the eventual fragmentation and subsequent switch table exhaustion. Consequently, we intend to assess and thoroughly evaluate the feasibility and computational complexity of specific procedures leveraging these two properties.

Controller performance and flexibility. The control plane is a crucial component of the BlueBridge system. As it primarily rests upon the underlying controller architecture, we require a mature, efficient, and reliable framework. Potential candidates under investigation include Beacon, Nettle, OpenDaylight, and ONOS. While ONOS [2] and OpenDaylight [16] are comprehensive frameworks with a multitude of features and industry support, they are comparably slow. [9] Beacon [5] and the Nettle language [22] feature extremely fast processing and response times, but are largely academic. Beacon is not under active development and Nettle is written in Haskell with unclear maturity and flexibility. BlueBridge is tightly coupled to the capabilities of the control platform and we have to weigh benefits of each control plane paradigm. Opting for the right framework is essential for the system to succeed.

4 PROOF OF CONCEPT SYSTEM

To evaluate the practicality of the aforementioned design we built a simple proof-of-concept tool operating entirely on IPv6 memory addressing. The implementation is minimal and focuses on appropriate conversion and network transport techniques to demonstrate the basic IPv6 shared memory use case and its benefits. The virtual memory system as well as the control plane and consistency management properties of the BlueBridge design are left as future work. The client-server program is written in C and Python and utilizes basic C memory operations.

4.1 System Aspects

The client consists of a library of simple remote allocate, read, write, and free functions which are mirrored on the server side. Allocation is chosen at random out of a list of possible host numbers. Instead of `mmap()`, the client stores acquired pointers in a simple list structure to read and write its data. When accessing remote memory the client inserts the IPv6 pointer into the header and sends it using a Linux datagram socket. The server retrieves the destination IP section, converts the 128 IPv6 address into a 64 bit pointer, and faithfully performs the requested action. We currently do not check for invalid or corrupted pointer addresses, which may lead to errors on the server side. A fully asynchronous, non-blocking client and server programs as well as checking pointer addresses are functions left for future work

4.2 Networking Aspects

We run our proof of concept system in Mininet [10], a networking virtualization engine primarily used for SDN research. Our current network setup consists of a single switch and controller managing up to 42 nodes. We are running a simple Open vSwitch which is pre-configured over several static `add-flow` commands which specify the appropriate server subnets. All forwarding is performed on layer two only and all flow entries are MAC-based. As these type of operations can be just as effectively covered by the static configuration, we have decided to not include the controller in this prototype.We leave controller integration to future work.

We still rely on the Linux network stack for communication, which automatically configures all necessary operations below transport layer. This includes neighbor discovery, packet construction, route entry management, and packet forwarding. Every pointer is a unique IPv6 address and hosts do not reply to unknown addresses. As a consequence, our system needs dedicated routing table entries for each node and utilize the network discovery protocol to function properly. As it is impossible to specify IPv6 subnet proxies natively, every node runs a NDP-proxy server which responds to NDP solicitation requests. For every pointer that is sent out into the network, a NDP solicitation request is broadcast. The proxy of the owning server will respond and the appropriate routing entry is created on all relevant nodes. This behaviour is highly undesirable as it generates round-trip overhead, nearly doubles network traffic, and may lead to peculiar problems (see Section 5.5.1). It is thus mandatory to abolish the dependence on NDP for our system to succeed. As it is a default feature of Linux transport layer operations, we are planning to move to raw socket programming entirely. This also provides us with the benefit of high customizability and the potential to define a leaner, specialized protocol.

4.3 IPv6 Remote Paging

We also created a `userfaultfd` program to perform remote paging, reminiscent of how DSM systems would transparently retrieve remote memory for an application. When a page fault occurs, the system determines if the page is remote or local. If the page is remote, it performs whatever communication is necessary to retrieve it, if it is local, it fetches the page. [21] In our implementation, we moved most of the `client` logic into `userfaultfd`. The program `mmaps` *n* number of addresses into memory marked to be handled by the userfaultfd handler. When it mmaps the memory, it allocates remote memory on the server and maintains a mapping of local addresses to remote addresses. Every access causes a page fault which is handled by our handler. We then forced page fault accesses to these pointers. For each page fault, the program checks to see if the pointer is remote or not by checking a hardcoded map. If the pointer is remote, it performs a READ operation on the remote memory and loads that into the local pointer.

5 EVALUATION

We evaluate our implementation in two ways. First, we look at the simulated speed up with a directory service switch to determine the overhead improvement our design brings. Next, we consider how the overhead compares to disk access if the communication used is RDMA.

5.1 Experimental Setup

As mentioned in Section 4, we used Mininet to simulate the network and switch for our tests. In our evaluation, we have three hosts, three servers, three NDP proxies, one switch, and one controller. Mininet was run on a machine with an Intel(R) Core(TM)2 Quad CPU Q9400 @ 2.66GHz, 4 GB of memory, and the mininet configuration reached a peak bandwidth of about 800 Mbits/sec. Each experiment was measured in nanoseconds, when the median latency and 95th percentile were calculated, the latencies were then converted into μs.

As mentioned before, this implementation uses a static switch configuration (learning switch). Although this makes our performance numbers not entirely representative of our proposed design, it does provide a good indication as to how the system will perform once it is fully built.

5.2 Remote Paging Latency

One of the main benefits of moving the management into the switch is the improved performance. This improved performance comes from reducing a roundtrip time (RTT) to the directory service. But this assumes that the RTT to a directory service incurs enough cost to affect the overall latency. To explore this, we simulated three scenarios: 1) shared memory system with a centralized directory service, 2) shared memory system with a directory service in the switch, and 3) a normal system disk access. We include the disk access in our measurements to provide a baseline for comparisons. We do not claim that we currently outperform a disk access, although our performance numbers point to the ability to do so with proper optimizations.

The shared memory system with a centralized directory service (Remote w/ DS) is simulated by querying the server for the IPv6 address of a piece of memory it requires to access. The server will then respond with an IPv6-compliant pointer for the client to use in a READ request. The shared memory with a directory service in the switch (Remote w/o DS) stores the IPv6-compliant pointer when allocating the memory, thus allowing it to skip the step of asking a directory service which machine to send its pointer to. This is not our exact design, as we would have a rule in the switch which recognizes which machine to send this pointer. It will be similar in overhead cost because it does not query the directory service. The normal system disk access (Local) reads a dummy file from disk and loads it into memory.

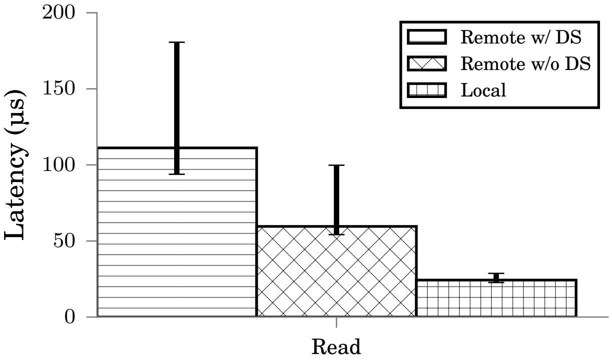

Figure 4: A comparison of the total latency per page fault of reads under three conditions: remote read with a directory service call, remote read without a directory service call, and accessing local disk. This is the median latency over 1,500 accesses of different pages with 95th percentile error bars.

We ran each program with 1,500 pages, where it accesses each page once. We record the latency at multiple points of the execution path to determine where each program spends most of its time as well as the total latency. We then graph the median of these latencies with the 95th percentile error bars. The total latency is shown in Figure 4 and appears as expected. The Remote w/ DS performs the worst and the disk access performs the best. We expected the disk access to perform the best because we did not optimize our proof of concept to use RDMA, therefore avoiding the server CPU. We discuss the implications of using RDMA in the next section.

We breakdown the total latency into five categories: enter handler, exit handler, access DS (directory service), get data, and other. Enter handler is the amount of time it takes from the application level access of a page which faults till the handler receives the event notification. Exit handler is the time it takes after the handler finishes its work till the application is no longer blocked. Access DS is the amount of time the handler spent querying the directory service for the machine to send the request to. Get data is the amount of time the handler spends getting the data from the remote host or disk.

The interesting part of the latency breakdown, Figure 5, is how much time the Access DS and Get data take compared to the other operations, they take approximately 9x longer. Based on that, it's obvious that reducing the execution path to one RTT instead of two is desirable. The reason the RTTs are drastically slower is the fact that we perform all communications over UDP on Ethernet. If these operations were done with RDMA, there should be a more significant speed up.

5.3 Potential Gains of RDMA

Most modern shared or remote memory systems leverage RDMA. Due to time constraints we could not leverage it for our proof of concept, but we still consider it in our performance evaluation. In

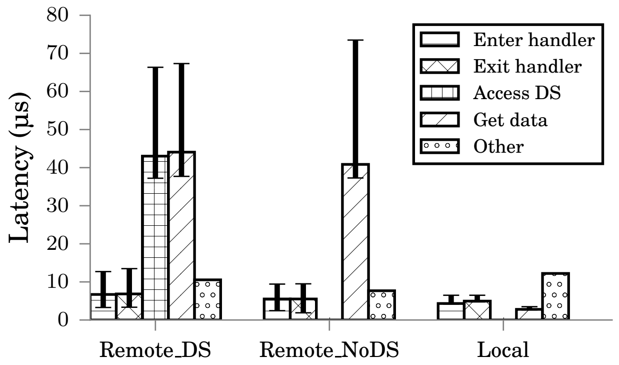

Figure 5: A breakdown of the latency cost for a read under three conditions: remote read with a directory service call, remote read without a directory service call, and accessing local disk. This is the median latency over 1,500 accesses of different pages with 95th percentile error bars.

our latency breakdown, we found that the RTT costs approximately 9x more than any of the other operations, which indicates that moving the directory service into the switch is very beneficial. But, we performed all communication in a non-optimal way, therefore we compare the amount of time it takes each remote operation with our communication method (UDP) and the equivalent RDMA method.

We have four operations, as stated in Section 3, CREATE, READ, UPDATE, DELETE. The RDMA equivalent for READ and UPDATE is RDMA read and RDMA write, respectively. CREATE and DELETE can be achieved using RDMA send/recv to issue commands to the remote server. According to experiments done by Ma et al. and Zhu et al., the approximate cost for an RDMA read or write operation is 3.4 μs and the approximate cost for an RDMA send/recv is 5.6 μs [15, 23].[1] We measured the latency of a small RDMA cluster using `qperf` to get one-way latencies for 4096 bytes. They mainly aligned with the numbers reported in [15, 23], so we use them in Table 1.

As seen in Table 1, using RDMA primitives instead of UDP improves the roundtrip time by 6x. Even with the improved communication latency, accessing a remote directory service incurs a high cost. In the worst case, it uses UDP and adds 45 μs, in the best case it uses RDMA primitives and adds 5.84 μs. It is still unnecessary overhead which can be mitigated by moving the directory service into the switch.

Another interesting observation of using RDMA, is that the theoretical latency would bring the total latency of our proposed solution into the same range as local disk. Our solution with the theoretical RDMA latency would cost approximately 22.116 μs, whereas the local disk latency is 24.304 μs.

	UDP (μs)	RDMA (μs)	Speed-up (x)
Create	34.919	5.84	5.979
Read	33.732	5.92	5.698
Update	34.151	5.96	5.730
Delete	32.963	5.84	5.644

Table 1: Comparison of our communication operations vs. their RDMA equivalent. For READ and UPDATE, the equivalents are RDMA read and write. For CREATE and DELETE, the equivalent is RDMA send/recv. Our latency medians were gathered over 1,500 runs of each operation. The RDMA measurements were gathered using `qperf`.

5.4 Data Migration

We attempted to perform data migration in our proof of concept system. Unfortunately, due to the NDP proxies required and the current network setup, data migration did not work. The switch was able to re-route the packets and the memory could easily be allocated at the same address on the new host server using `mmap()`, but the packets were getting dropped by the Linux network stack on the receiving server. This is because they were coming from an unknown address. Our system circumvents this issue by using the NDP proxies, but these do not work in the data migration case. Once we build a system which uses raw sockets and does not rely on NDP proxies, we will evaluate the data migration. We believe that data migration will work once the networking issues are fixed.

5.5 Observations

While running our tests we made several interesting observations.

5.5.1 NDP Table Exhaustion Attack. As we increased the number of pointers to access we noticed our program continuously deadlocking around the 1000th to 1500th iteration. The packet of the client never reached its destination. However, neither server nor client showed signs of exhaustion and race conditions were unlikely due to the strictly sequential nature of the program. We initially suspected the switch to overload and drop packets, but the device forwarded correctly. The actual fault occurred on the client side. Although the `send()` system call of the program returned correctly without error, the packet was never forwarded to the interface. The cause of this issue was the NDP neighbor table on the client host. The client accidentally performed a NDP Table Exhaustion Attack [18] (see Figure 6) on its own NDP neighbor table. The protocol generates a mapping for every unique IPv6 address to the destination interface. While the client allocates several thousand pointers, several thousand entries are created in the table. By default Ubuntu's NDP tables only have a size of around 4096 byte and the values expire slowly. The table is quickly exhausted and no entries can be created, which leads to a packet being quietly dropped and the program stalling. It seems that the only effective mitigation technique is to increase table size. NDP Neighbor tables do not seem offer netmask level matching nor is it possible explicitly specify a expiration time lower than 1 second. This is a crucial design flaw and further motivation to abolish NDP in our system. As we are moving towards raw socket messaging regardless, we are, as of this moment, not concerned about this problematic behavior.

[1] Reported latencies were one-way latencies for 2k and 4k pages. We doubled the reported latencies to make them roundtrip

```
[46768.440041] neighbour: ndisc_cache: neighbor table overflow!
[46770.564832] neighbour: ndisc_cache: neighbor table overflow!
[46770.580728] neighbour: ndisc_cache: neighbor table overflow!
```

Figure 6: NDP table exhaustion error messages.

5.5.2 *Mininet Performance Limits.*

Initially we planned to test the rack-scale scalability of our system on Mininet. However, we faced limitations regarding the physical resources of our test machines. We conducted the tests on a Ubuntu 16.04 LTS machine with 12GB of RAM and a four-core i5 760 @ 2.80GHz processor. The first test involved launching 42 hosts, each running a client/server process as well as ndp-proxy daemon. Each client performed a 100 full iterations, but the test did not complete as several clients froze

In the second test, a single client performed a 1000 full iterations. The client also froze. A 42 unit Mininet cluster significantly stresses the CPU of the host system, thus it is unclear whether these results are due to Mininet's scalability constraints or design flaws in our system. In addition, NDP seems to be a significant contributor to the load on the system. As every system maintains neighbor discovery, new pointers create solicitation storms across all nodes. We plan to retry these tests after implementing raw socket messaging.

5.5.3 *Disk Access Latency.*

The reported disk access latency numbers (Local in Figure 4) are much lower than previously reported disk access latencies in literature. This is due to the file being accessed from the buffer cache instead of the disk every read. Since the page fault handler reads the same file, at the offset of 0, the file's content is most likely stored in the buffer cache instead of causing a disk seek every access. We plan to re-run the tests in future work with a larger file, random offsets, and buffer cache flushing to ensure that the file is read from disk every page fault.

6 CONCLUSION

We have described the full system design and proof of concept implementation for BlueBridge, a network managed memory system. We have shown the feasibility of our design in a proof of concept system, then evaluated that system. The speedups gained by our proof of concept indicate that our full system implementation should produce promising results. As this was a proof of concept implementation, we have left work to be completed at a later date.

Future Work. One outstanding work item necessary to complete BlueBridge is the switch controller. We plan to implement a switch controller which handles most functionality attributed to a directory service, such as memory migration due to hot spots or failure, replication, etc. We plan to complete our data migration evaluation with this controller. If possible, we would like to also evaluate the failure recovery, replication, and any other features we get to implement. Next we plan on implementing concurrency control for the system to provide semantics around concurrent writes in this system. Lastly, we plan on implementing a library or API that an application can interface with to use BlueBridge.

REFERENCES

[1] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer* 29, 2 (Feb. 1996), 18–28. DOI:https://doi.org/10.1109/2.485843

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 1–6. DOI:https://doi.org/10.1145/2620728.2620744

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and others. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[4] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

[5] David Erickson. 2013. The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, New York, NY, USA, 13–18. DOI:https://doi.org/10.1145/2491185.2491189

[6] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. 1995. Implementing Global Memory Management in a Workstation Cluster. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 201–212. DOI:https://doi.org/10.1145/224057.224072

[7] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[8] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 202–215. DOI:https://doi.org/10.1145/2934872.2934908

[9] Masayoshi Kobayashi, Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan Van Reijendam, Paul Weissmann, and Nick Mckeown. 2014. Maturing of OpenFlow and Software-defined Networking Through Deployments. *Comput. Netw.* 61 (March 2014), 151–175. DOI:https://doi.org/10.1016/j.bjp.2013.10.011

[10] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. ACM, New York, NY, USA, Article 19, 6 pages. DOI:https://doi.org/10.1145/1868447.1868466

[11] Chung-Sheng Li, Hubertus Franke, Colin Parris, Bulent Abali, Mukil Kesavan, and Victor Chang. 2017. Composable architecture for rack scale big data computing. *Future Generation Computer Systems* 67 (2017), 180 – 193. DOI:https://doi.org/10.1016/j.future.2016.07.014

[12] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (Nov. 1989), 321–359. DOI:https://doi.org/10.1145/75104.75105

[13] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 31–44. http://dl.acm.org/citation.cfm?id=2930611.2930614

[14] S. Liang, R. Noronha, and D. K. Panda. 2005. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *2005 IEEE International Conference on Cluster Computing*. 1–10. DOI:https://doi.org/10.1109/CLUSTR.2005.347050

[15] S. Ma, J. Kim, and S. Moon. 2016. Exploring Low-Latency Interconnect for Scaling Out Software Routers. In *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*. 9–15. DOI:https://doi.org/10.1109/HIPINEB.2016.12

[16] J. Medved, R. Varga, A. Tkacik, and K. Gray. 2014. OpenDaylight: Towards a Model-Driven SDN Controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. 1–6. DOI:https://doi.org/10.1109/WoWMoM.2014.6918985

[17] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. *Grappa: A latency-tolerant runtime for large-scale irregular applications.* Technical Report. Technical Report UW-CSE-14-02-01, University of Washington.

[18] P. Nikander, J. Kempf, and E. Nordmark. 2004. *IPv6 Neighbor Discovery (ND) Trust Models and Threats.* RFC 3756. RFC Editor.

[19] B. Nitzberg and V. Lo. 1991. Distributed shared memory: a survey of issues and algorithms. *Computer* 24, 8 (Aug 1991), 52–60. DOI:https://doi.org/10.1109/2.84877

[20] Open Networking Foundation 2015. *Technical Specification: OpenFlow Specification 1.5.0* (0x06 ed.). Open Networking Foundation.

[21] J. Protic, M. Tomasevic, and V. Milutinovic. 1996. Distributed shared memory: concepts and systems. *IEEE Parallel Distributed Technology: Systems Applications* 4, 2 (Summer 1996), 63–71. DOI:https://doi.org/10.1109/88.494605

[22] Andreas Voellmy and Paul Hudak. 2011. Nettle: Taking the Sting out of Programming Network Routers. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages (PADL'11)*. Springer-Verlag, Berlin, Heidelberg, 235–249. http://dl.acm.org/citation.cfm?id=1946313.1946339

[23] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 523–536. DOI:https://doi.org/10.1145/2829988.2787484