Going Back to the Future with Camelot

Bronson Bouchard
Amanda Carbonari
Stewart Grant
Fabian Ruffy

ABSTRACT

Variety of applications require large scale data processing [4]. Many big data processing frameworks attempt to abstract away the complexities inherent with distributed data processing by enforcing programming models [5, 17, 26]. This leads to the frameworks imposing undo overhead on the applications and performing worse than a highly optimized single threaded laptop implementation [16].

We propose taking the simplicity of a single machine implementation and combining it with the advantages of a distributed implementation. We envision a system which turns a datacenter rack or equivalent cluster into a large NUMA machine. We realize the first part of this system in Camelot, a network managed distributed shared memory (DSM) backend.

Camelot is the next generation of a previous system, BlueBridge, which implemented the basis for network managed DSM. We expand upon BlueBridge to bring it closer to our envisioned system by adding support for multi-threading, improved paging policies, and fault tolerance. We evaluate Camelot on these three axises to assess the performance boon and functional capabilities of our extensions.

1 INTRODUCTION

As the amount of digital data collected increases, the need for data-intensive large scale processing increases. For example, Facebook processes over 500 terabytes a day and has invested in a large data processing pipleline to process that data [4]. Large scale data processing is not limited to large technology companies, different disciplines in the sciences have also taken advantage of computing power to make advancements. For example, high performance computing has been used to improve processing times of electron microscopy images and super computers have been used to implement and develop DNA sequencing systems [19, 23].

Previous work has looked at solving the issue of large scale data processing for different types of applications (i.e., graph processing, streaming, etc.). Distributed processing frameworks often have to make the trade off between performance and generality by enforcing certain abstractions or programming models on the developer [5, 13, 17]. This, unfortunately, restricts the types of applications which can run on these frameworks, often leading to developers attempting to force their workload into one of these frameworks. If a developer wishes to use Hadoop, they must write their algorithm in the MapReduce paradigm [5]. Or, if they wish to run graph processing on GraphLab, they must adhere to the vertex-centric processing paradigm [13].

Yet, it has been shown that many of these frameworks out performed by a single threaded laptop implementation [16]. McSherry, Isard, and Murray showed that his implementation of certain processing algorithms in RUST outperformed the most popular frameworks for that algorithm. But, programming highly performant code requires knowledge of the system (i.e., operating system tuning) and is not necessarily extensible (different tuning parameters per dataset and algorithm). The application developer needs to be able to reason about their algorithm but should not have to think about the underlying system to gain performance.

New trends in technology make it possible to have a combined approach, where a distributed framework abstracts away all complexity associated with distribution and presents the user with a single NUMA machine interface. We envision a system which turns a datacenter rack or equivalent cluster into a large NUMA machine. We realize the first part of this system in Camelot, the network managed distributed shared memory backend.

Distributed shared memory (DSM) is made realistic by faster interconnects. Traditionally, DSMs did not achieve adoption due to low throughput and high latency of network data transfers [12]. Now, with remote direct memory access (RDMA) and faster networks, the prospect of achieving low-latency (3 - 5 microsecond) is much more attainable. Due to this, there has been a resurgence of research into DSM systems and remote memory systems [6, 18, 20].

The recent work closest to Camelot, Grappa, exposes a single shared memory space with a simplified interface to allow for expressive but simple applications to run on the system. Although, both systems, our vision and Grappa, attempt to provide the same service (simple expressible interface on DSM), we provide optimizations and features beyond Grappa. We move memory management into the network to reduce the amount of network traffic and improve performance. Camelot also provides fault tolerant memory by adding different RAID levels to the system.

Camelot is the next generation of a previous system, BlueBridge, which implemented the basis for network managed DSM. We expand upon BlueBridge to bring it closer to our envisioned system by adding support for multi-threading, improved paging policies, and fault tolerance.

The contributions of this work are as follows:

- We added multi-threading support to BlueBridge, which allows for mulithreaded applications to take advantage of the DSM.
- We implemented different paging policies in Camelot to analyze the effectiveness of each policy to determine the best one for Camelot.
- We added fault tolerance by integrating a RAID manager into the client side page fault handler.

We evaluated each improvement to the system to determine the effects on performance. We found that our system is able to reliably process data which vastly exceed the local amount of memory. It and scales linearly in performance, but is yet unable to make proper use of network resources. Application runtime can be greatly improved by implementing fast running replacement policies and the choice of policy has big impact on performance. In memory RAID can significantly reduce memory usage (over 50% on common

configurations) with a computational overhead of ~6% compared to competing solutions [21].

Our paper will be organized as follows. First we will discuss previous work in the space in Section 2. Then we will discuss the design and implementation of Camelot in Section 3. Section 4 details the evaluation methodology and results. Finally, we conclude with discussion and future work.

2 RELATED WORK

There has been extensive work in shared and remote memory systems over the past decades. We break related work into two broad categories: distributed shared memory systems, fault tolerance in distributed shared memory systems, and systems which expose a single machine interface.

2.1 Distributed Shared Memory.

Distributed shared memory is an field of operating systems, which has been extensively studied over the decades. Two major reoccurring issues in shared memory are the performance impact of non-uniform memory accesses and consistency management. Often, these two aspects are at odds with each other. Systems of note, TreadMarks [9] and Cashmere [24], use different techniques to improve performance by trading off consistency. Treadmarks attempts to reduce the amount of communication necessary to enforce memory consistency. Similarly, Cashmere is a "two-level" shared memory system, which attempts to meet the challenge of performance by leveraging low-latency read-write capabilities. Camelot differs from both approaches in that we intend to attempt centralize shared memory management. Using a network addressing scheme, network elements are able to enforce consistency, reducing communication overhead and roundtrip time. the switch to reduce network overhead.

2.2 Fault Tolerance in DSM

Prior work on fault tolerant main memory in a distributed context was done by the RAMCloud project [21]. Their approach uses full replication a key-value store, similar to RAID1, and disks which provide a durable log for the system to recover from. Our approach to fault tolerant memory requires less memory, at the cost of computation. We implement RAID4/5 at the page level which requires that only a single replica contains parity data while supporting the same level of fault tolerance as RAMCloud.

2.3 Single Machine Interface.

This section looks at two DSMs which most closely resemble our system goals: a distributed framework which exposes a simple single machine interface to the developer. Grappa, unlike previous DSM work, relies on parallelism to achieve performance instead of relying on memory locality [18]. This similar to Camelot, as it does not rely on locality for performance, but they differ in the since that Camelot attempts to achieve that performance by moving management to the network and improving the network transfer times. Piccolo exposes a key-value table interface to computations running on different machines which acts as shared distributed state [22]. Piccolo is similar to the programming model we wish to use in Camelot, but we envision an even more general programming model. Piccolo requires the developer to organize their computation into application kernels while reasoning about how those kernels will be distributed [22]. Camelot, on the other hand, will allow applications to be written in a high-level, easy to understand language, such as Python, which provides more expressibility to the developer.

While entirely general we present Camelot in the context of a big data processing framework. Systems such as Hadoop [5], Pregal [14], Spark [27] use embarrassingly parallel constructs suchas MapReduce, or a *think like a vertex* programming mentality. These constructs allow for extreme scale but restrict programmers to their constructs and often produce inefficient algorithms. In contrast Camelot allows developers to process big data using conventional C. Efficient implementation of many algorithms can be achieved using data parallel techniques found in Dryad [8], Naiad [17], and TensorFlow [1]. These frameworks have proven notoriously hard for developers to understand, and debug in a distributed setting. Camelot's difficulty is exactly that of writing multithreaded C.

3 SYSTEM

Camelot is a functional extension of the BlueBridge system. In this section we describe BlueBridge, its limitations, and the extensions which compose the Camelot system; mainly multithreading, paging algorithms, and fault tolerance.

3.1 Prior System

Camelot is a successor to a simplistic DSM system. In the prior system, "BlueBride" all have an unrestricted view of the entire virtual address space. Client program were limited to a single thread, and all paging was performed using a naive FIFO paging strategy. BlueBride made use of a novel 128 bit IPv6 remote addressing mechanism which we adopted into Camelot. Remote addresses are represented as 128-bit IPv6-compatible pointers. On the client side, all remote addresses are stored as combination of the virtual pointer provided by the service node and its source IP address. If a client accesses remote memory, it inserts the address into the IP header of the request packet. The switch will route it automatically to the correct server node.

3.2 Network and Virtual Memory Stack

To support custom protocols without being dependent on the Linux networking stack, Camelot uses its own network library. The library is implemented using raw sockets, Berkely Packet Filters (BPFs) [15] and `packet mmap` [3]. At runtime, the application allocates a shared ring buffer, which it registers with the kernel. After binding the socket to an interface, the kernel copies any event into the receive buffer and continuously dispatches packets in the send buffer. To reduce the amount of packets inserted into the ring buffer, applications are able to insert a BPF byte-filter code, which directs the kernel to only copy relevant packets. This approach is substantially faster than conventional Linux send and receive, as it allows the removal of unnecessary procedures. Camelot also uses a cache management system to provide users with a section of managed memory. Users interfacing with the system initialize the virtual memory manager with a call that specifies cache and global address space size. The call will return a pointer to the managed cache

memory, behaving like a typical C `malloc()`. The global memory space is allocated via `mmap()` and completely protected. Faulting calls are subsequently mapped to the frame cache. This forms the page table of the system.

3.3 Page Eviction

BlueBridge's page eviction implementation was not sufficient for large scale distributed memory. Its inefficient FIFO implementation performed poorly with memory banks in excess of 4GB. We decided to not just fix the FIFO implementation but to experiment with other replacement policies to investigate optimal distributed paging.

Our objective in implementing these policies was O(1) insertion, retrieval, and deletion which was achieved by utilizing combined hashtable and list data structures. The implementation of the LFU cache is based off of an O(1) LFU algorithm from Kentan et al [10] in which a linked list of entries hold a use count and are each a parent of a linked list of entries containing the page. When a page is used it's position in the list data structure is found within the hashtable and the entry is moved to the next use count entries linked list. On eviction the tail entry of the head of the frequency list is removed.

We developed an LRU and improved FIFO paging strategy both use a linked list accompanied by a hashtable. The LRU version finds the entry associated with the page using the hashtable and moves it to the front of the list. On eviction the tail of the list is the least recently used and is evicted. The improved FIFO uses the hashtable in the same way as the LRU policy but only removes the tail entry and avoids moving used entries. The variety of policies gives some insight into what policy would be best and gives more options while running applications on Camelot. Camelot can efficiently handle the paging for large datasets loaded into memory.

3.4 Threading

As a distributed shared memory system Camelot must support multithreading, simply for multi-core support and the ability to explore sharing cache coherence semantics. In order to support multithreading, the network as well as virtual memory stack have to be thread-safe.

3.4.1 Threading in the Network library. Initially, threads were sharing a single ring buffer and individually consumed the packets placed by the kernel. Ideally, a thread would loop over the buffer, pick and discard relevant packets, and ignore packets of other threads. Frequently however, threads were not scheduled in time, causing their packets in the buffer to be displaced or dropped. The only apparent, immediate fix to this problem seems to be increasing the ring buffer size. In the final version of the system, only minimal sharing occurs. Each thread manages their own buffer which it has bound to the interface. To avoid unnecessary copy overhead thread register a packet filter and maintain their own UDP port.

3.4.2 Threading in the Virtual Memory System. In the virtual memory system, sharing, too, is minimal. Page metadata and the frame table are global, but sliced in subsets. Each thread owns a slice of the frame cache, which it has full control over. Threads are only able to access frames in their own cache, and are unable to evict or fill pages of other threads. While accessing and sharing pages of the global address table is theoretically possible, no support for coherence and consistency exists yet.

3.5 Fault Tolerance

Camelot achieves fault tolerance via in memory RAID. Thus far we have implemented 4 RAID variants 0,1,4,5 which provide each RAID's fault tolerant guarantees. RAID 0 is data striping only, and provides no fault tolerance. The failure of a single memory server causes applications to hang, this variant was developed as a baseline. RAID 1 is full replication, applications running RAID 1 can survive up to $n-1$ memory node failures. This RAID has the same guarantees as RAMCloud and was developed for the sake of comparison. RAID 4 & 5 use erasure encoding, in the form of parity to gain the tolerance of a single memory server failure. In RAID 4 parity is resident on a single machine, whereas RAID 5 distributes parity across all servers.

Camelots RAID manager resides in its page fault handler on client side applications. When a page fault occurs the RAID manager performs a RAID variant, as specified by a configuration file. In the case of RAID 0 with 4 memory servers the manager manager splits the evicted 4096 byte page into 1024 byte stripes and issues a separate async request to 4 servers. Upon receiving 4 acks the RAID manager returns control to the user level application. The same protocol holds for RAID 1, although full pages are broadcast to all servers. When a page is requested in RAID 0, all servers must respond with their stripe to compose a page, whereas RAID 1 returns control to applications after receiving its first replica.

RAID 4 and 5 require parity calculation on page evictions, and repair computation when failures occur. Parity calculation, and repair of 4K pages takes between 10 and 15 microseconds. The main bottleneck of these operations is cache memory bandwidth, as the calculation itself only requires the execution of C's XOR function. In the failure case when a page from a failed node is not received correction is run on the fly and repaired pages are passed to the application so it can operate opaque to failures. In such cases detecting that a page is missing via timeout is the dominant factor which increases latency. Our current implementation uses a fixed timeout of 100 microseconds, and no failure detector has been implemented as of yet. In the future using a dynamic timeout, or failure detector could dramatically decrease the cost of running repairs.

4 EVALUATION

We measured various performance aspects of Camelot to measure its scalability and competitiveness with existing data processing platforms. We developed a set of benchmark programs with diverse memory access patterns, to measure the quality of our page eviction strategies, threading performance, and raid overhead.

4.1 Experimental setup

Our tests were performed in the Mininet [11] as well as on real hardware. Mininet is a host-local network emulator intended to rapidly prototype SDN and data center network environments. Mininet allow us to verify and deploy code quickly, without having to worry about hardware constraints or configuration. While

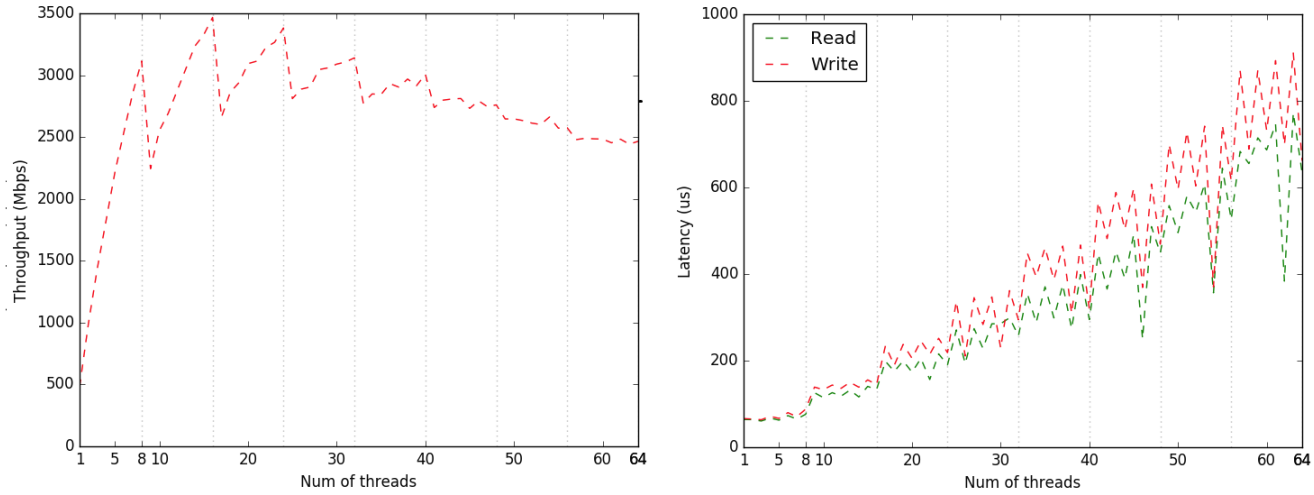

Figure 1: Microbenchmarks of the Camelot request API. The two figures show latency and throughput with threads pinned to cores in round-robin fashion.

absolute performance may not exactly be accurate, relative improvements are generally reliable. Our real hardware setup consisted of three servers, each of which is equipped with a 10-Gigabit X540-AT2 NIC, 32 GB of memory, and two four-core Intel(R) Xeon(R) E5-2407 v2 CPUs. The cores do not support hyperthreading. The microbenchmarks and threading tests were performed on real hardware to provide an accurate picture of the current system efficiency. The remaining tests were run in Mininet, as absolute performance was not of concern.

4.2 Threading Performance

To understand the implications of multithreading, and to verify the functionality of our current networking stack, we conducted several microbenchmarks. We pinned each thread to one core in a round-robin fashion, as this approach has proven itself to be more stable and reliable during testing. Our benchmark consisted of a million read and write requests, which we then scaled up to 64 threads. The results are shown in Fig 1. As expected, throughput scales linearly up to eight threads and latency does not increase. Unfortunately, the program is unable to exhaust the NIC bandwidth and only reaches around 3.5 out of 10 Gbit per second. At the same time, while latency in Mininet is on the order of 9-10 microseconds, real RTT is 60-70. This caused by a lack of request batching and the use of slow kernel networking code.

We decided to go beyond the expected scaling of eight cores and explore the system behaviour up to 64 threads. The pinned experiment ran stable, demonstrating a reasonable reliability. The throughput results exhibit a sawtooth pattern, with a local maximum occurring every eighth thread. This pattern is caused by the number of cores and pinning. However, this does not explain the maxima achieved by 16 and 24 threads. We assume that these may be caused by batching of requests in the NIC buffer, which is beneficial for throughput. Latency behaves as expected, with a mild increase in early iterations that transitions into wild instability the more threads are added.

4.3 Paging Performance

The purpose of our page performance evaluation was to quantify the performance improvement over the BlueBridge paging policy and to determine which paging policy is most efficient. Our first benchmark was running PageRank with varying amounts of local memory and each paging policy. The entire runtime is measured on the Google web graph dataset, roughly 75MB. The second benchmark was sorting a large number of random integers with varying local memory amounts.

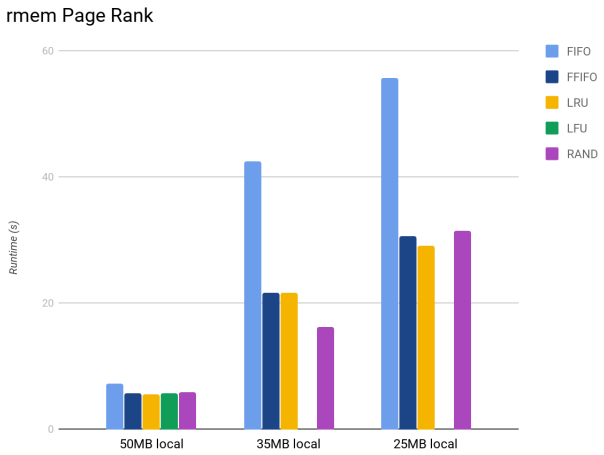

Figure 2: PageRank runtime benchmark for each policy with decreasing amount of local memory.

In the Pagerank benchmark, the LRU and improved FIFO policies outperformed the existing FIFO regardless of local memory size, achieving nearly half the execution time in the 35MB and 25MB tests as shown in Fig 2. As local memory was limited and more page replacements were made, the LFU eviction policy failed to replace the correct pages and execution time increased dramatically, the results of which for 35MB and 25MB local memory were 9 minutes and 40 minutes respectively. To test the effectiveness of the policies, a random replacement policy was implemented. Ideally there would have been a substantial performance benefit from using the new replacement policies over the random replacement but the random replacement policy performed the best in the 35MB local memory test and still performed reasonably well with 50MB and 25MB. This is could be due to the policies implemented being based only on write usage as the nodes are moved around on segfault. For further work it would be beneficial to capture the read usage in an efficient way for the LRU and LFU policies.

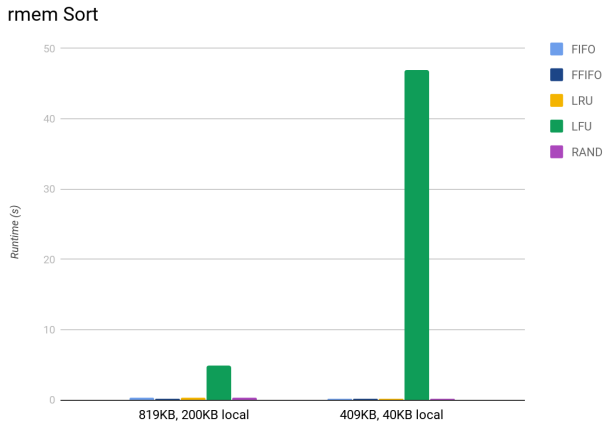

Figure 3: Sort runtime benchmark with decreasing amount of local memory for each policy.

In the Sort benchmark the existing FIFO and new replacement policies performed similarly with the exception of LFU. The results of the Sort benchmark in Fig 3 show the extreme difference between having a large number of local pages and a small number of local pages when using LFU.

4.4 RAID Overhead

Our goal in evaluating our RAID implementations is twofold. We first wished to establish a baseline overhead of running raid over regular remote memory, and to compare the cost of parity calculation relative to the no parity case common to RAMCloud. In both cases we found that the size of local cache used dramatically affected performance, so we report our performance results as a function of local cache size. We measured relative performance of these algorithms by timing iterations of a single threaded PageRank algorithm. Each iteration of page rank requires both a read and write step, so in each iteration all pages are read, and written to remote memory.

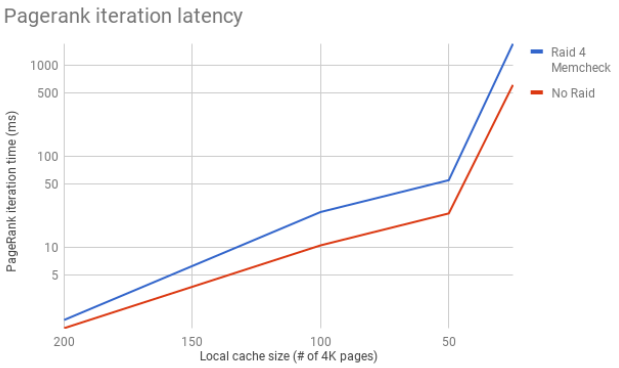

Figure 4: A logarithmic plot of RAID 4 running with memory check performed on reads, and parity calculations on writes, v.s traditional remote memory.

We found that running RAID4 with memory correction on a small local cache 1-2% of the entire graph) caused a 2x slowdown in PageRank iterations. The relation between performance and local cache size is plotted in Figure 4. As the number of local pages was increased to 10-20% of the entire graph the overhead was reduced to 1.5x slowdown. A large component of this overhead, is not the cost of performing XOR, it is due to our networking stacks lack of optimization for RAID. Our RAID manager is build directly on top of our remote memory stack, which issues single requests in the form of 4K pages. Therefore RAID 0,4,5 increase network bandwidth by $(n-1)x4K$ per page. Batching requests, and optimizing the network stack to read/write stripes of pages will substantially drop overhead.

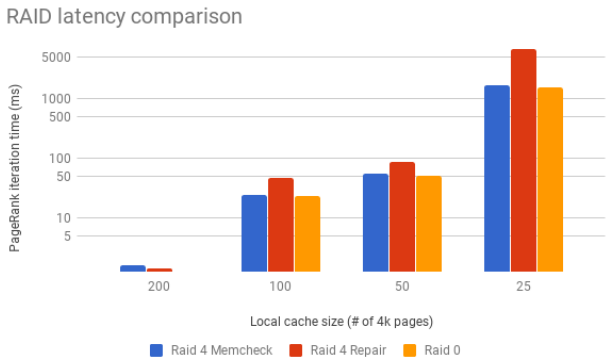

Figure 5: Raid 4, Raid 4 with failure, and Raid 0 vs local cache size

To evaluate the relative performance of RAID 4 & 5 compared to RAID 0 & 1 we likewise measured PageRank iteration times. We found that Raid 4 and 5 introduce a 6% overhead on both RAID 0 & 1, and that the overhead was directly proportional to parity

calculation time. Figure 5 plots our comparative results. RAID 0 & 1 ran competitively within 0.01% of each other. The only notable difference between then was the lack of a need for RAID 1 to wait for stragglers on reads. Given our setup the straggler problem was unobservable to our measurements. RAID 4 & 5 operate identically from a latency and bandwidth perspective. We also measured the performance penalty from running during a failure. We killed a single memory server while running PageRank and observer the overhead to be approximately 2x. The majority of the overhead was caused by waiting for a preset 100 microsecond timeout to fire and initiate the repair calculation. This overhead could be reduced by introducing a failure detector which would initiate repair immediately. In this case the overhead of running Camelot on 3 memory nodes under failure, vs RAMCloud is 6% while requiring 50% of the total memory.

5 DISCUSSION AND FUTURE WORK

In addition to a more concrete and sensible motivation, a substantial amount implementation and design work is needed to make Camelot a viable shared memory system.

5.1 Network Performance

Although benchmarking on the host-local Mininet emulator has shown that the code itself is sufficiently fast to achieve a request RTT of 9-10 microseconds, performance on real hardware is low. Multithreaded Camelot is not able to exhaust a single 10-Gbit NIC and experiences request latencies of roughly 50-60 microseconds. One thread averages throughput of 63 MByte per second, which pales against SSDs or even conventional HDDs. In order to make Camelot more competitive we have the following performance improvement goals:

5.1.1 Making use of userlevel network libraries. The Linux kernel negatively impacts the performance of highly specialized applications and frameworks. [7] Paying the cost of generality and kernel context-switches frequently adds an unacceptable overhead and inhibits applications from achieving their full performance potential. A viable alternative to relying on the default kernel network-stack is utilizing a userlevel customized high-performance library. Examples of such libraries included the Intel DPDK, ntop's PF-RING, and the netmap. [7] As a next step in development we plan to integrate one of the frameworks into our system, with the goal of achieving a sub-10 microseconds latency and full NIC exhaustion.

5.1.2 Event-based network handling. While the remote memory server does not currently seem to be the bottleneck, it is still singlethreaded and may become a future bottleneck. It may be good practice to develop a event-based processing framework which is multithreaded and scalable. This implementation may be trivial, as the server only processes basic CRUD requests. For high-performance, interesting options are either the same aforementioned userlevel networking libraries or the Express Data Path [25] framework, which fast-routes matching requests to our specific implementation.

5.2 Improving the virtual memory system

The managed virtual memory of Camelot is still far from optimal. Although we now have faster paging strategies, the memory system itself is vastly inefficient. Faults are processed over standard signal-handlers, which add a significant amount of latency to managed memory. In addition, we update eviction tracking information with every page-fault. We also do not track reads and writes on the cache itself and thus may use inaccurate information for eviction. To speed up the request trapping itself, we plan on adopting `userfaultfd` [2] as method of choice. It may reduce request latency and give us a more reliable and accurate sense of our virtual memory performance.

We are also considering a completely alternative path, in which we implement the system as a virtual disk or swap device. Instead of Camelot handling the virtual memory, swapping, and general management it would be the Linux operating system. If an application runs out of memory it will swap out to Camelot, just as it would swap to disk. This approach would alleviate the burden of reimplementing already existent and efficient approaches to cache-management and instead lets us focus on more novel systems contributions.

5.3 Tighter network integration

Our initial plan of involving the central TOR-switch was cut due to time-constraints. However, we believe that, partially because network elements can reason about requests, there is substantial synergy to be gained. For example, RAID requests may be automatically striped and consolidated on the switch side, making RAIDing fully transparent for clients. Similarly, it may be possible to deploy various load-balancing and migration schemes on the switch side, which save additional operation cost. As of now, this field of contribution remains an aspect of future work.

6 CONCLUSION

We have shown and discussed a new version of DSM which leverages in network management to expose a NUMA machine to the user. This provides a simple generic interface for the developer to program on, yet still provides the benefits of distribution of data and compute. We describe the Camelot system, which builds upon the BlueBridge system by adding multi-threading support, different paging policies, and RAID for memory fault tolerance. We evaluated each additions' performance impact and functionality. In our current setup, Camelot was able to scale up to eight cores, achieving a throughput of 3.5 Gbps with an average request latency of around 60 microseconds. Application runtime can be greatly improved by implementing fast running replacement policies and the choice of policy has big impact on performance. In memory RAID can significantly reduce memory usage (over 50% on common configurations) with a computational overhead of ~6% compaired to competing solutions [21].

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever,

K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. Arcangeli. userfaultfd, 2017. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt.

[3] U. Camara and J. Baudy. Packet mmap, 2017. https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt.

[4] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, 2016.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[6] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[7] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet io. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 29–38, Washington, DC, USA, 2015. IEEE Computer Society.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[9] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, 1994.

[10] P. Ketan, S. Anirban, and M. D. Matani. An o(1) algorithm for implementing the lfu cache eviction scheme, 2010.

[11] B. Lantz, B. Heller, and N. McKeown. 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, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4), 1989.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In *UAI*, 2010.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[16] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[18] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, 2015.

[19] J. O. Oelerich, L. Duschek, J. Belz, A. Beyer, S. D. Baranovskii, and K. Volz. Stemsalabim: A high-performance computing cluster friendly code for scanning transmission electron microscopy image simulations of thin specimens. *Ultramicroscopy*, 177, 2017.

[20] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[21] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[22] 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'10*, pages 293–306, 01 2010.

[23] M. J. Puckelwartz, L. L. Pesce, V. Nelakuditi, L. Dellefave-Castillo, J. R. Golbus, S. M. Day, T. P. Cappola, G. W. Dorn II, I. T. Foster, and E. M. McNally. Supercomputing for the parallelization of whole genome analysis. *Bioinfomatics*, 30.

[24] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. *ACM SIGOPS Operating Systems Review*, 31(5):170–183, 1997.

[25] I. Visor. Express data path (xdp), 2017. https://www.iovisor.org/technology/xdp.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.