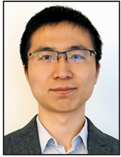
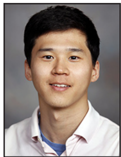
Decentralized Memory Disaggregation Over Low-Latency Networks

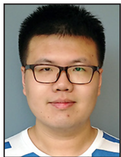
JUNCHENG GU, YOUNGMOON LEE, YIWEN ZHANG, MOSHARAF CHOWDHURY, AND KANG G. SHIN



Juncheng Gu is a PhD student at the University of Michigan. He is broadly interested in systems and networks. His current focus is resource disaggregation using RDMA networks. jcgu@umich.edu



Youngmoon Lee is a PhD candidate at the University of Michigan. He works on cloud and mobile systems, and his current research focuses on resilient cloud computing and resource management. ymoonlee@umich.edu



Yiwen Zhang is a master's student at the University of Michigan. His research interests include computer networks and RDMA performance isolation. yiwenzhg@umich.edu

Mosharaf Chowdhury is an Assistant Professor in the EECS Department at the University of Michigan. His research ranges from resource disaggregation in low-latency RDMA networks to geo-distributed analytics over the WAN, with a common theme of enabling application-infrastructure symbiosis across different layers of corresponding software and hardware stacks. mosharaf@umich.edu

Memory disaggregation can expose remote memory across a cluster to local applications. However, existing proposals call for new architectures and/or new programming models, making them infeasible. We have developed a practical memory disaggregation solution, Infiniswap, which is a remote memory paging system for clusters with low-latency, kernel-bypass networks such as RDMA. Infiniswap opportunistically harvests and transparently exposes unused memory across the cluster to unmodified applications by dividing the swap space of each machine into many chunks and distributing them to unused memory of many remote machines. For scalability, it leverages the power of many choices to perform decentralized memory chunk placements and evictions. Applications using Infiniswap receive large performance boosts when their working sets are larger than their physical memory allocations.

Motivation

Modern operating systems (OSes) provide each application with a virtual memory address space that is much larger than its physical memory allocation. Whenever an application addresses a virtual address whose corresponding virtual page does not reside in the physical memory, a *page fault* is raised. If there is not enough space in the physical memory for that virtual page, the virtual memory manager (VMM) may need to *page out* one or more in-memory pages to a block device, which is known as the *swap space*. Subsequently, the VMM brings the missing page into the physical memory from the swap space; this is known as *paging in*.

Performance Degradation from Paging

Due to the limited performance of traditional swap spaces—typically, rotational hard disks—paging in and out can significantly affect application performance. To illustrate this issue, we select four memory-intensive applications: (1) a standard TPC-C benchmark running on the VoltDB in-memory database; (2) two Facebook-like workloads running on the Memcached key-value store; (3) the TunkRank algorithm running on PowerGraph with a Twitter data set; and (4) GraphX running the PageRank algorithm in Apache Spark using the same Twitter data set.

We run each application in its own container with different memory constraints. *x*% in the X-axes of Figure 1 refers to a run inside a container that can hold at most *x*% of the application's working set in memory, and ×< 100 forces paging in from/out to the machine's *swap space*.

Figure 1 shows significant, non-linear impact on application performance due to paging. For example, a 25% reduction of memory results in a 5.5× and 2.1× throughput loss for VoltDB and Memcached, respectively; PowerGraph and GraphX worsen marginally. However, another 25% reduction makes VoltDB, Memcached, PowerGraph, and GraphX up to 24×, 17×, 8×, and 23× worse, respectively. These gigantic performance degradations reflect the potential benefits that an efficient memory disaggregation system can deliver.

Kang G. Shin is the Kevin & Nancy O'Connor Professor of Computer Science in the Department of Electrical Engineering and Computer Science, University of Michigan. His current research focuses on QoS-sensitive computing and networking as well as on embedded real-time and cyber-physical systems. He has supervised the completion of 80 PhDs and has authored/co-authored more than 900 technical articles, a textbook, and more than 30 patents or invention disclosures; he has received numerous best paper awards. He was a co-founder of a couple of startups and also licensed some of his technologies to industry. kgshin@umich.edu

Characteristics of Memory Imbalance

Memory utilization is imbalanced across machines in a cluster. Although some machines are under heavy memory pressure, others in the same cluster can still have unused memory. Causes of imbalance include placement and scheduling constraints [3, 4] and resource fragmentation during packing [8]. To understand the presence of memory imbalance in clusters and corresponding opportunities, we analyzed traces from two production clusters: (1) a 3000-machine data analytics cluster (Facebook) and (2) a 12,500-machine cluster (Google) running a mix of diverse short- and long-running applications.

Presence of Imbalance. We measured memory utilization imbalance by calculating the 99th-percentile to the median usage ratio over 10-second intervals (Figure 2). With a perfect balance, these values would be 1. However, we found this ratio to be 2.40 in Facebook and 3.35 in Google more than half the time; meaning, most of the time, more than half of the cluster aggregate memory remains unutilized.

Temporal Variabilities. Although skewed, memory utilizations remained stable over short intervals, which is useful for predictable decision-making when selecting remote machines. We observed that average memory utilizations of a machine remained stable for smaller durations with very high probabilities. For the most unpredictable machine in the Facebook cluster, the probabilities that its current memory utilization from any instant will not change by more than 10% for the next 10, 20, and 40 seconds were 0.74, 0.58, and 0.42, respectively. For Google, the corresponding numbers were 0.97, 0.94, and 0.89, respectively.

The presence of memory imbalance and its temporal variabilities suggest opportunities for harvesting unused memory across a cluster by memory disaggregation.

Infiniswap Overview

Infiniswap is a decentralized memory disaggregation solution for clusters with low-latency, kernel-bypass networks such as RDMA. The main goal of it is to *efficiently expose all of a cluster's memory to user applications*. To avoid modifying existing applications or OSes, Infiniswap provides remote memory to local applications through the already-existing paging mechanism.

Infiniswap has two primary components—Infiniswap block device and Infiniswap daemon—that are present in every machine and work together without any central coordination (Figure 3).

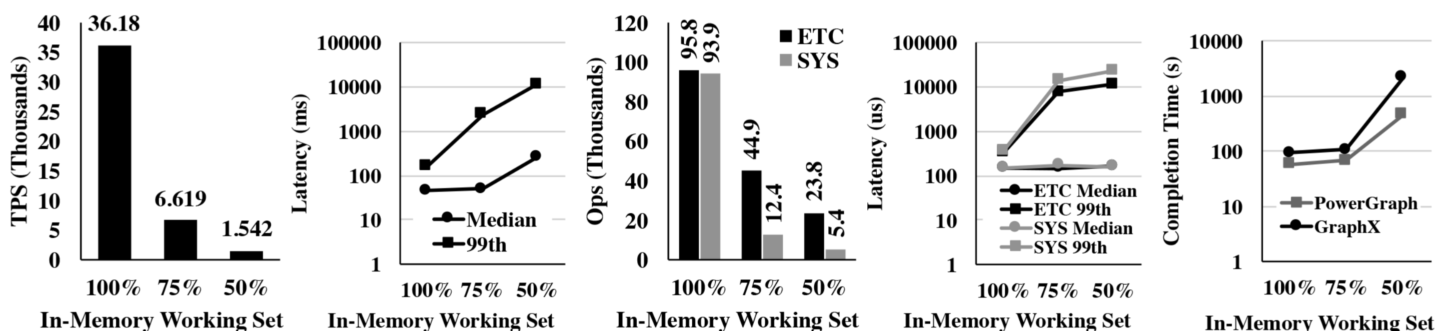


Figure 1: For modern in-memory applications, a decrease in the percentage of the working set that fits in memory often results in a disproportionately larger loss of performance. This effect is further amplified for tail latencies. All plots show single-machine performance and the median value of five runs. Lower is better for the latency-related plots (lines), and the opposite holds for the ones (bars). Note the logarithmic Y-axes in the throughout-related latency/completion time plots.

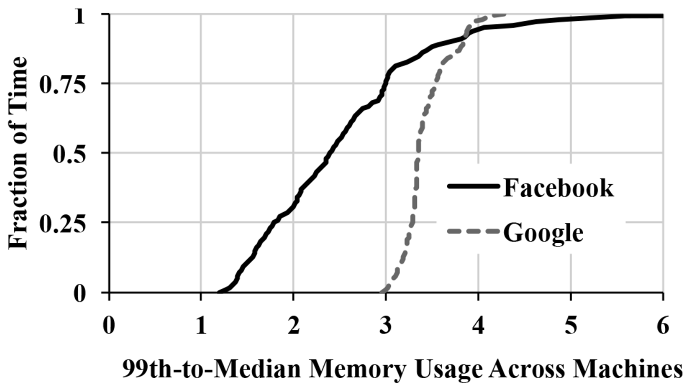


Figure 2: Imbalance in 10s-averaged memory usage in two large production clusters at Facebook and Google

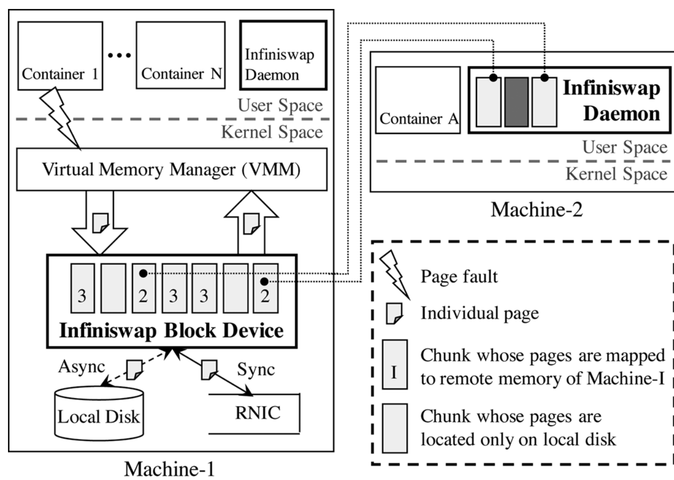


Figure 3: Infiniswap architecture. Each machine loads a block device as a kernel module (set as swap device) and runs an Infiniswap daemon. The block device divides its storage space into chunks and transparently maps them across many machines' remote memory; paging happens at page granularity via RDMA.

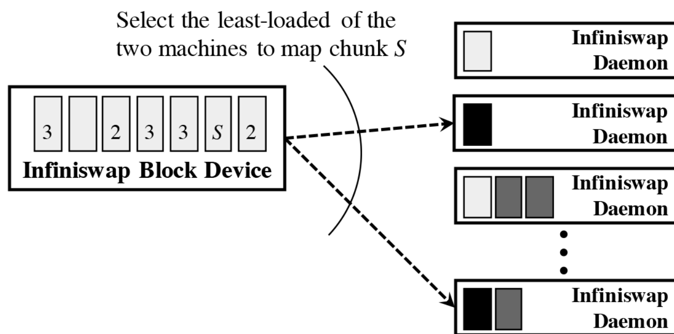


Figure 4: Infiniswap block device uses power of two choices to select machines with the most available memory. It prefers machines without any of its chunks over those that have chunks. In this way, its chunks can be distributed across as many machines as possible.

The Infiniswap block device exposes a conventional block device I/O interface to the virtual memory manager (VMM), which treats it as a fixed-size swap partition. The entire storage space of this device is logically partitioned into fixed-size *chunks* ("ChunkSize"). A chunk represents a contiguous region, and it is the unit of remote mapping and load balancing in Infiniswap. Chunks from the same block device can be mapped to multiple remote machines' memory for load balancing. The VMM stores and retrieves data from the Infiniswap block device at page granularity. All pages belonging to the same chunk are mapped to the same remote machine. On the Infiniswap daemon side, a chunk is a physical memory region of ChunkSize that is mapped to and used by an Infiniswap block device as remote memory.

Infiniswap consults the status of remote memory mapping to handle paging requests. If a chunk is mapped to remote memory, Infiniswap synchronously writes a page-out request for that chunk to remote memory using `RDMA WRITE`, while writing it asynchronously to the local disk. If it is not mapped, Infiniswap synchronously writes the page only to the local disk. For page-in requests, Infiniswap reads data from the appropriate source; it uses `RDMA READ` for remote memory.

The Infiniswap daemon only participates in control plane activities. It (1) responds to chunk-mapping requests from Infiniswap block devices; (2) pre-allocates its local memory when possible to minimize time overheads in chunk-mapping initialization; and (3) proactively evicts chunks, when necessary, to ensure minimal impact on local applications. All control plane communications take place using `RDMA SEND/RECV`.

Scalability. Infiniswap leverages the well-known power of choices techniques [6, 7] during both chunk placement and eviction. The reliance on decentralized techniques makes Infiniswap more scalable by avoiding the need for constant coordination, while still achieving low-latency mapping and eviction.

Fault Tolerance. With the decentralized approach, Infiniswap does not have a single point of failure. It considers unreachability of remote daemons (e.g., due to machine failures, daemon process crashes, etc.) as the primary failure scenario. If a remote daemon becomes unreachable, the Infiniswap block device relies on the remaining remote memory and the local backup disk. If the local disk also fails, Infiniswap provides the same failure semantic as of today.

Efficient Memory Disaggregation via Infiniswap Block Device

An Infiniswap block device logically divides its entire storage space into multiple chunks of fixed size (ChunkSize). Using a fixed size throughout the cluster simplifies chunk placement and eviction algorithms and their analyses.

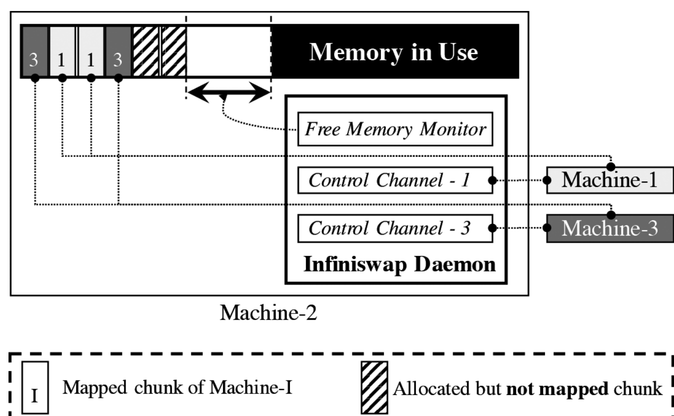


Figure 5: The Infiniswap daemon periodically monitors available free memory to pre-allocate chunks and to perform fast evictions. Each machine runs one daemon.

Remote Chunk Placement

Each chunk starts in the *unmapped* state. Infiniswap monitors the paging activity rates of each chunk using an exponentially weighted moving average (EWMA). When the paging activity of an unmapped chunk crosses the HotChunk threshold, Infiniswap attempts to map that chunk to a remote machine's memory.

The chunk placement algorithm has the following goals. First, it should *distribute chunks from the same block device* across as many remote machines as possible in order to minimize the impacts of future evictions from (or failures of) remote machines. Second, it attempts to *balance memory utilization* across all the machines to minimize the probability of future evictions. Finally, it must be *decentralized* to provide low-latency mapping without central coordination.

Instead of randomly selecting an Infiniswap daemon without central coordination, we leverage the power of two choices [6] to minimize memory imbalance across machines (Figure 4). First, Infiniswap divides all the machines (M) into two sets: those that already have any chunk of this block device (M_{old}) and those that do not (M_{new}). Next, it contacts two Infiniswap daemons and selects the one with the lowest memory usage. It first selects from M_{new} and then, if necessary, from M_{old}. The two-step combination distributes chunks across many machines while decreasing load imbalance in a decentralized manner.

Handling Chunk Evictions and Remote Failures

Upon receiving an eviction message from the Infiniswap daemon, the Infiniswap block device marks the chunk as unmapped. All future requests of the unmapped chunk will go to disk. The Infiniswap block device cannot send the eviction response back to the Infiniswap daemon until all the in-flight requests of that chunk are completed. The workflow of handling remote failures

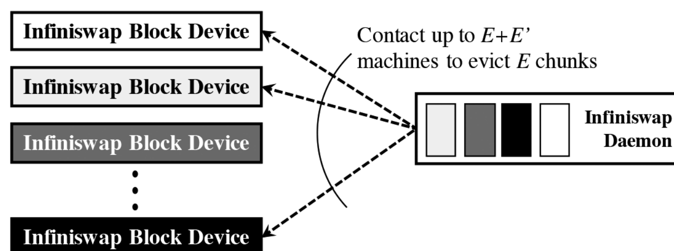


Figure 6: The Infiniswap daemon employs batch eviction (i.e., contacting *E´* more chunks to evict *E* chunks) for fast eviction of *E* lightly active chunks.

is similar to that of chunk eviction: mark the affected chunk(s) as unmapped, and forward future requests to disk.

Transparent Remote Memory Reclamation via Infiniswap Daemon

The core functionality of each Infiniswap daemon is to claim memory on behalf of remote block devices as well as reclaiming them on behalf of the applications on its host.

Memory Management

The Infiniswap daemon periodically monitors the total memory usage of everything else running on its host. In order to be transparent to applications on the same machine, it focuses on maintaining a "HeadRoom" amount of free memory in the machine by controlling its own total memory allocation. The optimal value of "HeadRoom" should be dynamically determined based on the amount of memory and the applications running in each machine. Our current implementation does not include this optimization and uses 8-GB "HeadRoom" by default on 64-GB machines.

When the amount of free memory grows above "HeadRoom," the Infiniswap daemon proactively allocates chunks of size ChunkSize and marks them as unmapped. Proactive allocation of chunks makes the initialization process faster when an Infiniswap block device attempts to map to that chunk; the chunk is marked mapped at that point.

When free memory shrinks below "HeadRoom," the Infiniswap daemon proactively releases chunks in two stages. It starts by releasing unmapped chunks. Then, if necessary, it *evicts E* mapped chunks.

Decentralized Chunk Eviction

To minimize the performance impact on the Infiniswap block devices that are remotely mapped, the Infiniswap daemon should select the least-active mapped chunks for eviction. The key challenge arises from the one-sided `RDMA (READ/WRITE)` operations used in the data plane of Infiniswap. While this avoids CPU involvements, it also prevents the Infiniswap

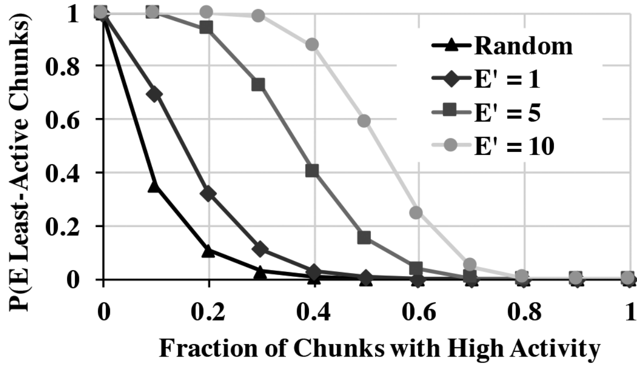


Figure 7: Analytical eviction performance for evicting *E(= 10)* chunks for varying values of *E´*. Random refers to evicting *E* chunks one by one uniformly randomly.

daemon from gathering any paging activities of the mapped chunks without first communicating with the corresponding block devices.

Consider a scenario where a daemon needs to release E mapped chunks. At one extreme, the solution is to collect global knowledge by contacting *all* related block devices to determine the least-used E chunks. This is prohibitive when E is significantly smaller than the total number of mapped chunks. Having a centralized controller would not have helped either, because this would require *all* Infiniswap block devices to frequently report their chunk activities.

At the other extreme, one can randomly pick one chunk at a time without any communication. However, in this case, the likelihood of evicting a busy chunk is very high. Consider a parameter $p_b \in [0, 1]$, and assume that a chunk is busy (i.e., it is experiencing paging activities beyond a fixed threshold) with probability p_b. The probability of finding E lightly active chunks would be $(1 - p_b)^E$. As the cluster becomes busier (p_b increases), this probability plummets (Figure 7).

Batch Eviction. Instead of randomly evicting chunks without any communication, we perform bounded communication to leverage generalized power of choices [7].

For E chunks to evict, the Infiniswap daemon considers $E + E'$ chunks, where $E' \leq E$. Upon communicating with the Infiniswap block devices of those $E + E'$ chunks, it evicts E least-active ones. The probability of finding E lightly active chunks in this case is

$$\sum_{E}^{E+E'} (1-p_b)^i {p_b}^{E+E'-i} \binom{E+E'}{i}$$

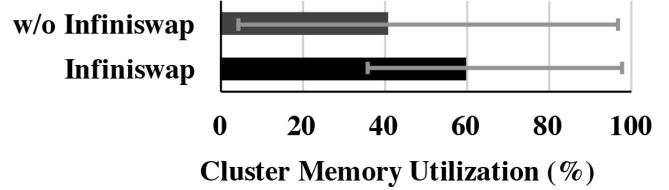


Figure 8: Using Infiniswap, memory utilization increases and memory imbalance decreases significantly. Error bars show the maximum and the minimum utilization across machines.

Figure 7 plots the effectiveness of batch eviction for different values of E' for $E = 10$. Even for moderate cluster load, the probability of evicting lightly active chunks is significantly higher using batch eviction.

Implementation

We have implemented Infiniswap as a loadable kernel module for Linux 3.13.0 and beyond in about 3500 lines of C code. Our block device implementation is based on nbdX [1], a network block device over Accelio framework, developed by Mellanox. Infiniswap daemons are implemented and run as userspace programs. More implementation details can be found in our NSDI paper [5].

Evaluation

We evaluated Infiniswap on a 32-machine, 56 Gbps Infiniband cluster on CloudLab [2] and highlight two key results as follows:

- In comparison to traditional swap spaces such as rotational disks, Infiniswap improves throughputs of unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark from 4× to up to 15.4× and tail latencies by up to 61×.
- Infiniswap benefits hold in a distributed setting. It increases cluster memory utilization by 1.47× using a small amount of network bandwidth.

The rest of this section describes how Infiniswap performs in a cluster with many applications. Details about our experimental setup, workload configurations, and more evaluation results can be found in our NSDI paper [5].

Cluster-Wide Performance

Methodology

We used the same application-workload combinations in Figure 1 to create about 90 containers. Each combination has an equal number of containers. About 50% of them had no memory constraint, close to 30% used the 75% memory constraint, and the rest used the 50% memory constraint. They were placed randomly across 32 machines to create a memory imbalance scenario similar to those shown in Figure 2.

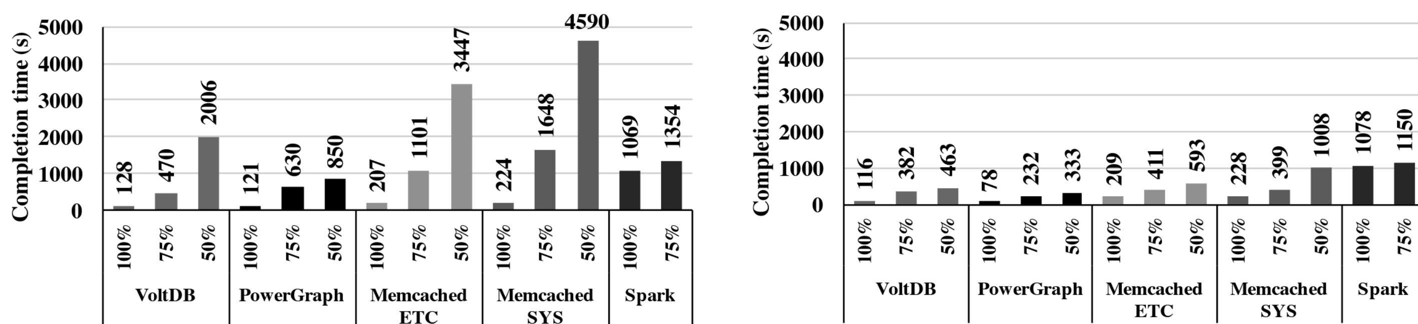


Figure 9: Median completion times of containers for different configurations in the cluster experiment. Infiniswap's benefits translate well to a larger scale in the presence of high application concurrency.

Cluster Memory Utilization

Infiniswap improves total cluster memory utilization by 1.47× by increasing it to 60% on average from 40.8% (Figure 8). Moreover, Infiniswap significantly decreases memory imbalance: the maximum-to-median utilization ratio decreased from 2.36× to 1.60×, and the maximum-to-minimum utilization ratio decreased from 22.5× to 2.7×.

Application-Level Performance

We observe that Infiniswap holds its benefits in the presence of cluster dynamics of many applications (Figure 9). Although improvements are sometimes lower than those observed in controlled single-instance scenarios [5], Infiniswap still provides 3×–6× improvements over disk for the 50% memory constraint.

Ongoing Efforts

We are actively extending Infiniswap in two directions:

Fault Tolerance

Infiniswap can tolerate the failures of remote machines with its backup disk. However, backing up data on hard disk becomes the performance bottleneck of the entire system when many swap bursts come together. We are considering trying to achieve the fault tolerance feature by distributing data to multiple remote machines using erasure coding.

Performance Isolation

Infiniswap provides remote memory to all the applications running on the machine. As such, it cannot distinguish between pages from specific applications. Swap requests originating from different applications share the same resources in Infiniswap, such as dispatch buffers in Infiniswap and cache on RDMA NICs. Consequently, Infiniswap cannot guarantee performance isolation among multiple applications on the same host.

Conclusion

Infiniswap is a pragmatic solution for memory disaggregation without requiring any modifications to applications, OSes, or hardware. It bypasses CPU through one-sided RDMA operations in the data plane for performance, and it uses scalable, decentralized remote memory placement and eviction schemes in the control plane for fault tolerance and scalability. We have demonstrated Infiniswap's advantages in substantially improving the performance of multiple popular memory-intensive applications. Infiniswap also increases the overall memory utilization of a cluster, and its benefits hold at scale.

The source code of Infiniswap and more information are available at https://infiniswap.github.io/infiniswap/.

Acknowledgments

Special thanks go to the entire CloudLab team—especially Robert Ricci, Leigh Stoller, and Gary Wong—for pooling together enough resources to make Infiniswap experiments possible. We would also like to thank the anonymous reviewers and our shepherd, Mike Dahlin, for their insightful comments and feedback that helped improve the paper. This work was supported in part by National Science Foundation grants CCF-1629397, CNS-1563095, CNS-1617773, by the ONR grant N00014-15-1-2163, and by an Intel grant on low-latency storage systems.

References

[1] Accelio-based network block device: https://github.com/accelio/NBDX.

[2] CloudLab: https://www.cloudlab.us.

[3] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica, "Surviving Failures in Bandwidth-Constrained Datacenters," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '12)*, pp. 431–442: http://bit.ly/2xDVSOs.

[4] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging Endpoint Flexibility in Data-Intensive Clusters," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '13)*, pp. 231–242: http://bit.ly/2fqyzgU.

[5] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient Memory Disaggregation with Infiniswap," i n *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*: http://bit.ly/2yFiMDl.

[6] M. Mitzenmacher, A. W. Richa, and R. Sitaraman, "The Power of Two Random Choices: A Survey of Techniques and Results," *Handbook of Randomized Computing* (Springer, 2001), pp. 255–312.

[7] G. Park, "Brief Announcement: A Generalization of Multiple Choice Balls-into-Bins," in *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC '11)*, pp. 297–298.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys '15)*: http://bit.ly/2ye4ecV.

Save the Date!

FAST '18

16th USENIX Conference on File and Storage Technologies

February 12–15, 2018 • Oakland, CA, USA

FAST '18 brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; everything from low-level storage devices to information management is of interest. The conference will consist of technical presentations, including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

The full program and registration will be available in December 2017.

www.usenix.org/fast18

Save the Date!

nsdi '18

15th USENIX Symposium on Networked Systems Design and Implementation

April 9–11, 2018 • Renton, WA, USA

NSDI '18 focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

The full program and registration will be available in January 2018.

www.usenix.org/nsdi18