Efficient Memory Disaggregation with INFINISWAP

Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin, *University of Michigan*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Efficient Memory Disaggregation with INFINISWAP

Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, Kang G. Shin
University of Michigan

Abstract

Memory-intensive applications suffer large performance loss when their working sets do not fully fit in memory. Yet, they cannot leverage otherwise unused remote memory when paging out to disks even in the presence of large imbalance in memory utilizations across a cluster. Existing proposals for memory disaggregation call for new architectures, new hardware designs, and/or new programming models, making them infeasible.

This paper describes the design and implementation of INFINISWAP, a remote memory paging system designed specifically for an RDMA network. INFINISWAP opportunistically harvests and transparently exposes unused memory to unmodified applications by dividing the swap space of each machine into many slabs and distributing them across many machines' remote memory. Because one-sided RDMA operations bypass remote CPUs, INFINISWAP leverages the power of many choices to perform decentralized slab placements and evictions.

We have implemented and deployed INFINISWAP on an RDMA cluster without any modifications to user applications or the OS and evaluated its effectiveness using multiple workloads running on unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark. Using INFINISWAP, throughputs of these applications improve between $4\times$ ($0.94\times$) to $15.4\times$ ($7.8\times$) over disk (Mellanox nbdX), and median and tail latencies between $5.4\times$ ($2\times$) and $61\times$ ($2.3\times$). INFINISWAP achieves these with negligible remote CPU usage, whereas nbdX becomes CPU-bound. INFINISWAP increases the overall memory utilization of a cluster and works well at scale.

1 Introduction

Memory-intensive applications [18, 20, 23, 77] are widely used today for low-latency services and data-intensive analytics alike. The main reason for their popularity is simple: as long as requests are served from memory and disk accesses are minimized, latency decreases and throughput increases. However, these applications experience rapid performance deteriorations when their working sets do not fully fit in memory (§2.2).

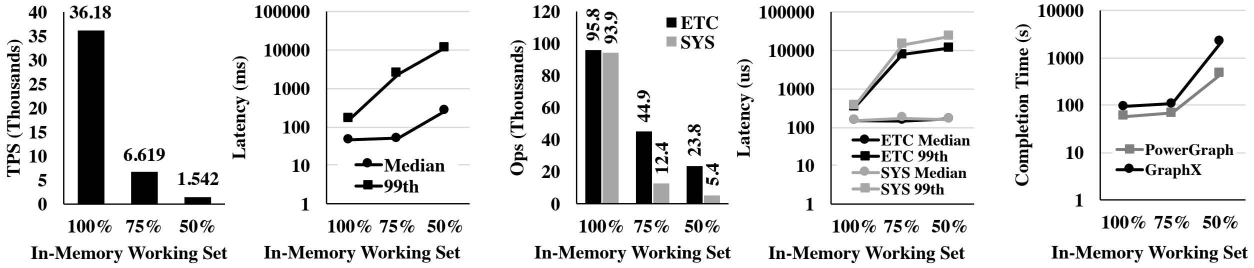
There are two primary ways of mitigating this issue: (i) rightsizing memory allocation and (ii) increasing the effective memory capacity of each machine. Rightsizing is difficult because applications often overestimate their requirements [71] or attempt to allocate for peak usage [28], resulting in severe underutilization and unbalanced memory usage across the cluster. Our analysis of two large production clusters shows that more than 70% of the time there exists severe imbalance in memory utilizations across their machines (§2.3).

Proposals for memory disaggregation [42, 49, 53, 70] acknowledge this imbalance and aim to expose a global memory bank to all machines to increase their effective memory capacities. Recent studies suggest that modern RDMA networks can meet the latency requirements of memory disaggregation architectures for numerous in-memory workloads [42, 70]. However, existing proposals for memory disaggregation call for new architectures [11, 12, 49], new hardware designs [56, 57], and new programming models [63, 69], rendering them infeasible.

In this paper, we present INFINISWAP, a new scalable, decentralized remote memory paging solution that enables efficient memory disaggregation. It is designed specifically for RDMA networks to perform remote memory paging when applications cannot fit their working sets in local memory. It does so *without* requiring any coordination or modifications to the underlying infrastructure, operating systems, and applications (§3).

INFINISWAP is not the first to exploit memory imbalance and disk-network latency gap for remote memory paging [2, 25, 31, 37, 40, 41, 55, 58, 64]. However, unlike existing solutions, it does not incur high remote CPU overheads, scalability concerns from central coordination to find machines with free memory, and large performance loss due to evictions from, and failures of, remote memory.

INFINISWAP addresses these challenges via two primary components: a block device that is used as the swap space and a daemon that manages remotely accessible memory. Both are present in every machine and work together without any central coordination. The INFINISWAP block device writes synchronously to remote memory for low latency and asynchronously to disk for fault-tolerance (§4). To mitigate high recovery overheads of disks in the presence of remote evictions and failures, we divide its address space into fixed-size slabs and place them across many machines' remote memory. As a result, remote evictions and failures only affect the perfor-



(a) TPC-C on VoltDB

(b) Facebook workloads [26] on Memcached

(c) Analytics

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 throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.

mance of a fraction of its entire address space. To avoid coordination, we leverage power of two choices [62] to find remote machines with available free memory. All remote I/O happens via RDMA operations.

The INFINISWAP daemon in each machine monitors and preallocates slabs to avoid memory allocation overheads when a new slab is mapped (§5). It also monitors and proactively evicts slabs to minimize performance impact on local applications. Because swap activities on the hosted slabs are transparent to the daemon, we leverage power of many choices [68] to perform batch eviction without any central coordination.

We have implemented INFINISWAP on Linux kernel 3.13.0 (§6) and deployed it on a 56 Gbps, 32-machine RDMA cluster on CloudLab [5]. We evaluated it using multiple unmodified memory-intensive applications: VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark using industrial benchmarks and production workloads (§7). Using INFINISWAP, throughputs improve between 4× (0.94×) and 15.4× (7.8×) over disk (Mellanox nbdX [2]), and median and tail latencies by up to 5.4× (2×) and 61× (2.3×), respectively. Memory-heavy workloads experience limited performance difference during paging, while CPU-heavy workloads experience some degradation. In comparison to nbdX, INFINISWAP does not use any remote CPU and provides a 2×–4× higher read/write bandwidth. INFINISWAP can recover from remote evictions and failures while still providing higher application-level performance in comparison to disks. Finally, its benefits hold in the presence of high concurrency and at scale, with a negligible increase in network bandwidth usage.

Despite its effectiveness, INFINISWAP cannot transparently emulate memory disaggregation for CPU-heavy workloads such as Spark and VoltDB (unlike memory-intensive Memcached and PowerGraph) due to the inherent overheads of paging (e.g., context switching). We still consider it useful because of its other tangible benefits. For example, when working sets do not fit in memory, VoltDB's performance degrades linearly using INFINISWAP instead of experiencing a super-linear drop.

We discuss related work in Section 9.

2 Motivation

This section overviews necessary background (§2.1) and discusses potential benefits from paging to remote memory in memory-intensive workloads (§2.2) as well as opportunities for doing so in production clusters (§2.3).

2.1 Background

Paging. Modern operating systems (OSes) support virtual memory to provide applications with larger address spaces than physically possible, using fixed-size pages (typically 4KB) as the unit of memory management. Usually, there are many more virtual pages than physical ones. Page faults occur whenever an application addresses a virtual address, whose corresponding page does not reside in physical memory. Subsequently, the virtual memory manager (VMM) consults with the page table to bring that page into memory; this is known as *paging in*. To make space for the new page, the VMM may need to *page out* one or more already existing pages to a block device, which is known as the *swap space*.

Any block device that implements an expected interface can be used as a swap space. INFINISWAP is written as a virtual block device to perform this role.

See [14] for a detailed description of memory management and its many optimizations in a modern OS.

Application Deployment Model. We consider a container-based application deployment model, which is common in production datacenters [28, 71, 76] as well as in container-as-a-service (CaaS) models [4, 8, 13, 50]. These clusters use resource allocation or scheduling algorithms [28, 43, 47, 73] to determine resource shares of different applications and deploy application processes

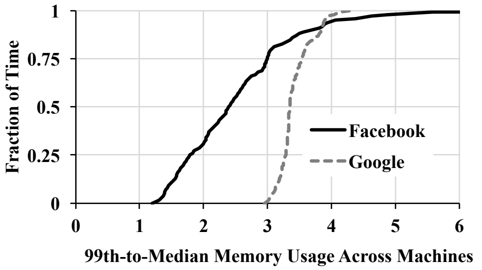


Figure 2: Imbalance in 10s-averaged memory usage in two large production clusters at Facebook and Google.

in containers to ensure resource isolation. Applications start paging when they require more memory than the memory limits of their containers.

Network Model. INFINISWAP requires a low-latency, RDMA network, but we do not make any assumptions about specific RDMA technologies (e.g., Infiniband vs. RoCE) or network diameters. Although we evaluate INFINISWAP in a small-scale environment, recent results suggest that deploying RDMA (thus INFINISWAP) on large datacenters may indeed be feasible [48, 61, 79].

2.2 Potential Benefits

To illustrate the adverse effects of paging, we consider four application types: (i) a standard TPC-C benchmark [22] running on the VoltDB [23] in-memory database; (ii) two Facebook-like workloads [26] running on the Memcached [18] key-value store; (iii) PowerGraph [45] running the TunkRank algorithm [1] on a Twitter dataset [52]; and (iv) PageRank running on Apache Spark [77] and GraphX [46] on the Twitter dataset. We found that Spark starts thrashing during paging and does not complete in many cases. We defer discussion of Spark to Section 7.2.4 and consider the rest here.

To avoid externalities, we only focus on single-server performance. Peak memory usage of each of these runs were around 10GB, significantly smaller than the server's total physical memory. We run each application inside containers of different memory capacities: $x\%$ 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 $x < 100$ forces paging. Section 7.2 has more details on the experimental setups.

We highlight two observations that show large potential benefits from INFINISWAP. First, paging has significant, non-linear impact on performance (Figure 1). For example, a 25% reduction in in-memory working set results in a $5.5\times$ and $2.1\times$ throughput loss for VoltDB and Memcached; in contrast, PowerGraph and GraphX worsen marginally. However, another 25% reduction makes VoltDB, Memcached, PowerGraph, and GraphX up to $24\times$, $17\times$, $8\times$, and $23\times$ worse, respectively.

Second, paging implications are highlighed particularly at the tail latencies. As working sets do not fit into memory, the 99th-percentile latencies of VoltDB and Memcached worsen by up to $71.5\times$ and $21.5\times$, respectively. In contrast, their median latencies worsens by up to $5.7\times$ and $1.1\times$, respectively.

These gigantic performance gaps suggest that a theoretical, 100%-efficient memory disaggregation solution can result in huge benefits, assuming that everything such solutions may require is ensured. It also shows that bridging some of these gaps by a practical, deployable solution can be worthwhile.

2.3 Characteristics of Memory Imbalance

To understand the presence of memory imbalance in modern clusters and corresponding opportunities, we analyzed traces from two production clusters: (i) a 3000-machine data analytics cluster (Facebook) and (ii) a 12500-machine cluster (Google) running a mix of diverse short- and long-running applications.

We highlight two key observations – the presence of memory imbalance and its temporal variabilities – that guide INFINISWAP's design decisions.

Presence of Imbalance. We found that the memory usage across machines can be substantially unbalanced in the short term (e.g., tens of seconds). Causes of imbalance include placement and scheduling constraints [28, 44] and resource fragmentation during packing [47, 76], among others. 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 2.4 in Facebook and 3.35 in Google more than half the time; meaning, most of the time, more than a 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. To analyze the stability of memory utilizations, we adopted the methodology described by Chowdhury et al. [32, §4.3]. Specifically, we consider a machine's memory utilization $U_t(m)$ at time t to be stable for the duration T if the difference between $U_t(m)$ and the average value of $U_t(m)$ over the interval $[t, t+T)$ remains within 10% of $U_t(m)$. 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. We believe

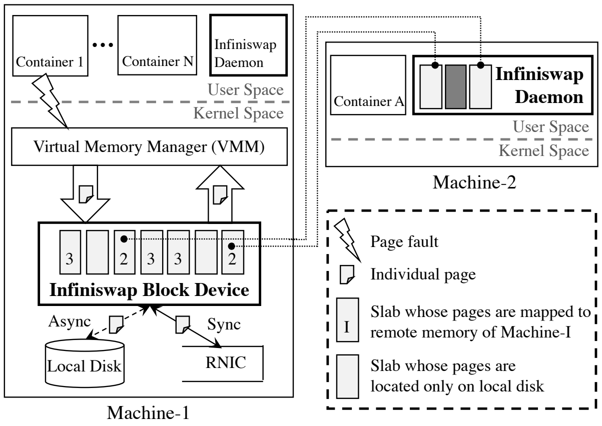


Figure 3: INFINISWAP architecture. Each machine loads a block device as a kernel module (set as swap device) and runs an INFINISWAP daemon. The block device divides its address space into slabs and transparently maps them across many machines' remote memory; paging happens at page granularity via RDMA.

that the higher probabilities in the Google cluster are due to its long-running services, whereas the Facebook cluster runs data analytics with many short tasks [24].

3 INFINISWAP Overview

INFINISWAP is a decentralized memory disaggregation solution for RDMA clusters that opportunistically uses remote memory for paging. In this section, we present a high-level overview of INFINISWAP to help the reader follow how INFINISWAP performs efficient and fault-tolerant memory disaggregation (§4), how it enables fast and transparent memory reclamation (§5), and its implementation details (§6).

3.1 Problem Statement

The main goal of INFINISWAP is to *efficiently expose all of a cluster's memory to user applications* without any modifications to those applications or the OSes of individual machines. It must also be *scalable, fault-tolerant, and transparent* so that application performance on remote machines remains unaffected.

3.2 Architectural Overview

INFINISWAP consists of two primary components – INFINISWAP block device and INFINISWAP daemon – that are present in every machine and work together without any central coordination (Figure 3).

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 address space of this device is logically partitioned into fixed-size *slabs* (SlabSize). A slab is the unit of remote mapping and load balancing in INFINISWAP. Slabs from the same device can be mapped to multiple remote machines' memory for performance and load balancing (§4.2). All pages belonging to the same slab are mapped to the same remote machine. On the INFINISWAP daemon side, a slab is a physical memory chunk of SlabSize that is mapped to and used by an INFINISWAP block device as remote memory.

If a slab is mapped to remote memory, INFINISWAP synchronously writes a page-out request for that slab 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 or reads, INFINISWAP consults the slab mapping to read from the appropriate source; it uses RDMA READ for remote memory.

The INFINISWAP daemon runs in the user space and only participates in control plane activities. Specifically, it responds to slab-mapping requests from INFINISWAP block devices, preallocates its local memory when possible to minimize time overheads in slab-mapping initialization, and proactively evicts slabs, when necessary, to ensure minimal impact on local applications. All control plane communications take place using RDMA SEND/RECV.

We have implemented INFINISWAP as a loadable kernel module for Linux 3.13.0 and deployed it in a 32-machine RDMA cluster. It performs well for a large variety of memory-intensive workloads (§7.2).

Scalability. INFINISWAP leverages the well-known power-of-choices techniques [62, 68] during both slab placement in block devices (§4.2) and eviction in daemons (§5.2). 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. Because INFINISWAP does not have a central coordinator, it does not have a single point of failure. If a remote machine fails or becomes unreachable, INFINISWAP relies on the remaining remote memory and the local backup disk (§4.5). If the local disk also fails, INFINISWAP provides the same failure semantic as of today.

4 Efficient Memory Disaggregation via INFINISWAP Block Device

In this section, we describe how INFINISWAP block devices manage their address spaces (§4.1), perform decentralized slab placement to ensure better performance and load balancing (§4.2), handle I/O requests (§4.3), and minimize the impacts of slab evictions (§4.4) and remote failures (§4.5).

4.1 Slab Management

An INFINISWAP block device logically divides its entire address space into multiple slabs of fixed size (SlabSize).

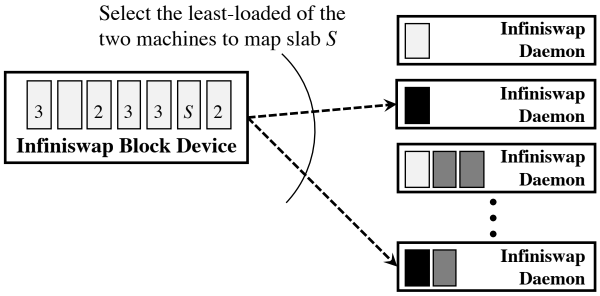


Figure 4: INFINISWAP block device uses power of two choices to select machines with the most available memory. It prefers machines without any of its slabs to those who have to distribute slabs across as many machines as possible.

Using a fixed size throughout the cluster simplifies slab placement and eviction algorithms and their analyses.

Each slab starts in the *unmapped* state. INFINISWAP monitors the page activity rates of each slab using an exponentially weighted moving average (EWMA) with one second period:

$$A_{\text{current}}(s) = \alpha \, A_{\text{measured}}(s) + (1 - \alpha) \, A_{\text{old}}(s)$$

where α is the smoothing factor ($\alpha = 0.2$ by default) and $A(s)$ refers to total page-in and page-out activities for slab s (initialized to zero).

When $A_{\text{current}}(s)$ crosses a threshold (HotSlab), INFINISWAP initiates remote placement (§4.2) to map the slab to a remote machine's memory. This late binding helps INFINISWAP avoid unnecessary slab mapping and potential memory inefficiency. We set HotSlab to 20 page I/O requests/second. In our current design, pages are not proactively moved to remote memory. Instead, they are written to remote memory via RDMA WRITE on subsequent page-out operations.

To keep track of whether a page can be found in remote memory, INFINISWAP maintains a bitmap of all pages. All bits are initialized to zero. After a page is written out to remote memory, its corresponding bit is set. Upon failure of a remote machine where a slab is mapped or when a slab is evicted by the remote INFINISWAP daemon, all the bits pertaining to that slab are reset.

In addition to being evicted by the remote machine or due to remote failure, INFINISWAP block devices may preemptively remove a slab from remote memory if $A_{\text{current}}(s)$ goes below a threshold (ColdSlab). Our current implementation does not use this optimization.

4.2 Remote Slab Placement

When the paging activity of an unmapped slab crosses the HotSlab threshold, INFINISWAP attempts to map that slab to a remote machine's memory.

The slab placement algorithm has multiple goals. First, it must *distribute slabs from the same block device* across as many remote machines as possible in order to minimize the impacts of future evictions from (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.

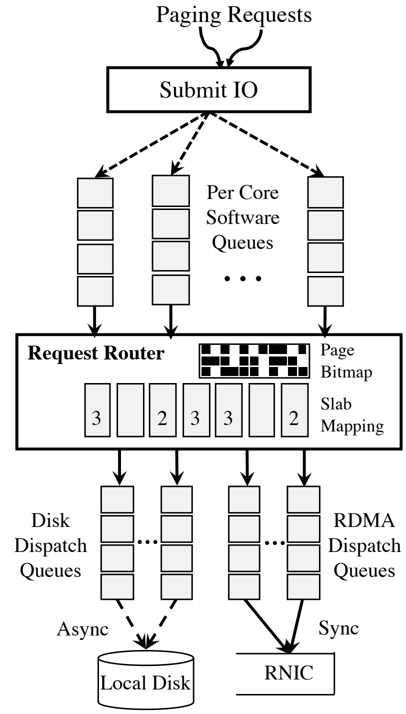


Figure 5: INFINISWAP block device overview. Each machine uses one block device as its swap partition.

One can select an INFINISWAP daemon uniformly randomly without central coordination. However, this is known to cause load imbalance [62, 68].

Instead, we leverage power of two choices [62] to minimize memory imbalance across machines. First, INFINISWAP divides all the machines ($\mathbb{M}$) into two sets: those who already have any slab of this block device ($\mathbb{M}_{\text{old}}$) and those who do not ($\mathbb{M}_{\text{new}}$). Next, it contacts two INFINISWAP daemons and selects the one with the lowest memory usage. It first selects from $\mathbb{M}_{\text{new}}$ and then, if required, from $\mathbb{M}_{\text{old}}$. The two-step combination distributes slabs across many machines while decreasing load imbalance in a decentralized manner.

4.3 I/O Pipelines

The VMM submits page write and read requests to INFINISWAP block device using the block I/O interface (Figure 5). We use the multi-queue block IO queuing mechanism [15] in INFINISWAP. Each CPU core is configured with an individual software staging queue, where block (page) requests are staged. The request router consults the slab mapping and the page bitmap to determine how to forward them to disk and/or remote memory.

Each RDMA dispatch queue has a limited number of entries, each of which has a registered buffer for RDMA communication. Although the number of RDMA dispatch queues is the same as that of CPU cores, they do not follow one-to-one mapping. Each request from a core is assigned to a random RDMA dispatch queue by hashing its address parameter to avoid load imbalance.

Page Writes. For a page write, if the corresponding slab is mapped, a write request is duplicated and put into both RDMA and disk dispatch queues. The content of the page is copied into the buffer of RDMA dispatch entry, and the buffer is shared between the duplicated requests. Once the RDMA WRITE operation completes, the page write is completed and its corresponding physical memory can be reclaimed by the kernel without waiting for the disk write. However, the RDMA dispatch entry – and its buffer – will not be released until the completion of the disk write operation. When Infiniswap cannot get a free entry from all RDMA dispatch queues, it blocks until one is released.

For unmapped slabs, a write request is only put into the disk dispatch queue; in this case, Infiniswap blocks until the completion of the write operation.

Page Reads. For a page read, if the corresponding slab is mapped and the page bitmap is set, an RDMA READ operation is put into the RDMA dispatch queue. When the READ completes, Infiniswap responds back. Otherwise, Infiniswap reads it from the disk.

Multi-Page Requests. To optimize I/O requests, the VMM often batches multiple page requests together and sends one multi-page (batched) request. The maximum batch size in the current implementation of Infiniswap is 128 KB (i.e., 32 4 KB pages). The challenge in handling multi-page requests arises in cases where pages cross slab boundaries, especially when some slabs are mapped and others are unmapped. In these cases, Infiniswap waits until operations on all the pages in that batch have completed in different sources; then, it completes the multi-page I/O request.

4.4 Handling Slab Evictions

The decision to evict a slab (§5.2) is communicated to a block device via the *EVICT* message from the corresponding Infiniswap daemon. Upon receiving this message, the block device marks the slab as unmapped and resets the corresponding portion of the bitmap. All future requests will go to disk.

Next, it waits for all the in-flight requests in the corresponding RDMA dispatch queue(s) to complete, polling every 10 microseconds. Once everything is settled, Infiniswap responds back with a *DONE* message.

Note that if $A(s)$ is above the HotSlab threshold, Infiniswap will start remapping the slab right away. Otherwise, it will wait until $A(s)$ crosses HotSlab again.

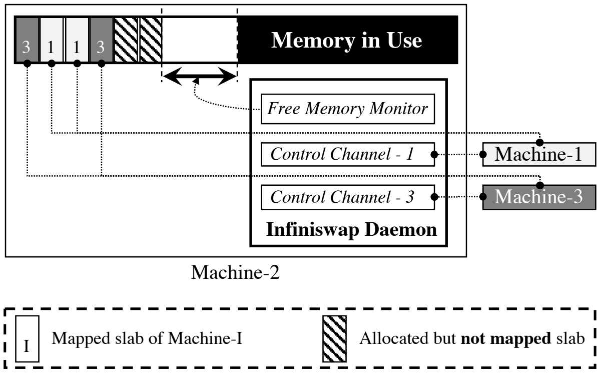


Figure 6: Infiniswap daemon periodically monitors the available free memory to pre-allocate slabs and to perform fast evictions. Each machine runs one daemon.

4.5 Handling Remote Failures

Infiniswap uses reliable connections for all communication and considers unreachability of remote Infiniswap daemons (e.g., due to machine failure, daemon process crashes, etc.) as the primary failure scenario. Upon detecting a failure, the workflow is similar to that of eviction: the block device marks the slab(s) on that machine as unmapped and resets the corresponding portion(s) of the bitmap.

The key difference and a possible concern is handling in-flight requests, especially *read-after-write* scenarios. In such a case, the remote machine fails after a page (P) has been written to remote memory but before it is written to disk (i.e., P is still in the disk dispatch queue). If the VMM attempts to page P in, the bitmap will point to disk, and a disk read request will be added to the disk dispatch queue. Because all I/O requests for the same slab go to the same disk dispatch queue, such read requests will be served by the on-disk data written by the previous write operation.

In the current implementation, Infiniswap does not handle transient failures separately. A possible optimization would be to use a timeout before marking the corresponding slabs unmapped.

5 Transparent Remote Memory Reclamation via Infiniswap Daemon

In this section, we describe how Infiniswap daemons (Figure 6) monitor and manage memory (§5.1) and perform slab evictions to minimize remote and local performance impacts (§5.2).

5.1 Memory Management

The core functionality of each Infiniswap daemon is to claim memory on behalf of remote Infiniswap block devices as well as reclaiming them on behalf of local applications. To achieve this, the daemon monitors the total

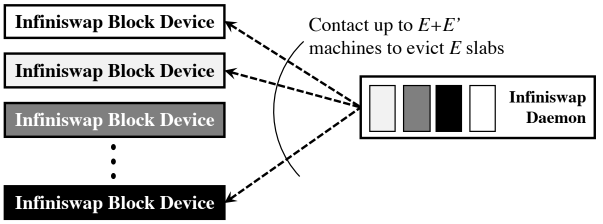


Figure 7: INFINISWAP daemon employs batch eviction (i.e., contacting E' more slabs to evict E slabs) for fast eviction of E lightly active slabs.

memory usage of everything else running on the machine using an EWMA with one second period:

$$U_{\text{current}} = \beta\, U_{\text{measured}} + (1 - \beta)\, U_{\text{old}}$$

where β is the smoothing factor ($\beta = 0.2$ by default) and U refers to total memory usage (initialized to 0).

Given the total memory usage, INFINISWAP daemon focuses on maintaining a HeadRoom amount of free memory in the machine by controlling its own total memory allocation at that point. 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, INFINISWAP proactively allocates slabs of size SlabSize and marks them as unmapped. Proactive allocation of slabs makes the initialization process faster when an INFINISWAP block device attempts to map to that slab; the slab is marked mapped at that point.

5.2 Decentralized Slab Eviction

When free memory shrinks below HeadRoom, INFINISWAP daemon proactively releases slabs in two stages. It starts by releasing unmapped slabs. Then, if necessary, it *evicts* E mapped slabs as described below.

Because applications running on the local machine do not care which slabs are evicted, when INFINISWAP must evict, it focuses on minimizing the performance impact on the machines that are remotely paging. The key challenge arises from the fact that remote INFINISWAP block devices directly interact with their allocated slab(s) via RDMA READ/WRITE operations without any involvement of INFINISWAP daemons. While this avoids CPU involvements, it also prevents INFINISWAP from making any educated guess about performance impact of evicting one or more slabs without first communicating with the corresponding block devices.

This problem can be stated formally as follows. *Given S mapped slabs, how to release the E least-active ones to leave more than* HeadRoom *free memory?*

At one extreme, the solution is simple with global knowledge. INFINISWAP daemon can contact *all* block devices in the cluster to determine the least-used E slabs and evict them. This is prohibitive when E is significantly smaller than the total number of slabs in the cluster. Having a centralized controller would not have helped either, because this would require all INFINISWAP block devices to frequently report their slab activities.

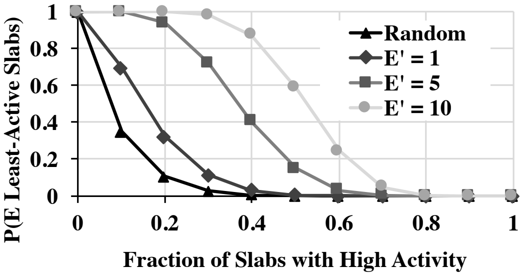


Figure 8: Analytical eviction performance for evicting $E(= 10)$ slabs for varying values of E'. Random refers to evicting E slabs one-by-one uniformly randomly.

At the other extreme, one can randomly pick one slab at a time without any communication. However, in this case, the likelihood of evicting a busy slab is very high. Consider a parameter $p_b \in [0, 1]$, and assume that a slab is busy (i.e., it is experiencing paging activities beyond a fixed threshold) with probability p_b. If we now reformulate the problem to *finding E lightly active slabs* instead of the least-active ones, the probability would be $(1 - p_b)^E$. As the cluster becomes busier (p_b increases), this probability plummets (Figure 8).

Batch Eviction. Instead of randomly evicting slabs without any communication, we perform bounded communication to leverage generalized power of choices [68]. Similar techniques had been used before for task scheduling and input selection [67, 75].

For E slabs to evict, INFINISWAP daemon considers $E + E'$ slabs, where $E' \le E$. Upon communicating with the machines hosting those $E + E'$ slabs, it evicts E least-active ones (i.e., the E slabs of $E + E'$ with the lowest $A(.)$ values). The probability of finding E lightly active slabs in this case is $\sum_{E}^{E+E'} (1 - p_b)^i {p_b}^{E+E'-i} \binom{E+E'}{i}$.

Figure 8 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 slabs are significantly higher using batch eviction.

The actual act of eviction is initiated when the daemon sends *EVICT* messages to corresponding block devices. Once a block device completes necessary bookkeeping (§4.4), it responds with a *DONE* message. Only then INFINISWAP daemon releases the slab.

6 Implementation

INFINISWAP is a virtual block device that can be used as a swap partition, for example, `/dev/infiniswap0`.

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, a network block device over Accelio framework, developed by Mellanox[2]. We also rely on stackbd [21] to redirect page I/O requests to the disk to handle possible remote failures and evictions. INFINISWAP daemons are implemented and run as user-space programs.

Control Messages. INFINISWAP components use message passing to transfer memory information and memory service agreements. There are eight message types. Four of them are used for placement and the rest (e.g., *EVICT*, *DONE*) are used for eviction.

A detailed list of messages, along with how they are used during placement and eviction, and corresponding sequence diagrams can be found in Appendix B.1.

Connection Management. INFINISWAP uses reliable connections for all communications. It uses one-sided RDMA READ/WRITE operations for data plane activities; both types of messages are posted by the block device. All control plane messages are transferred using RDMA SEND/RECV operations.

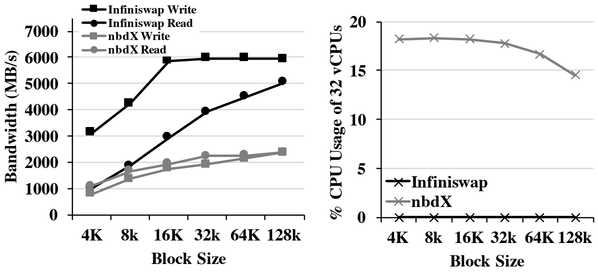
INFINISWAP daemon maintains individual connections for each block device connected to it instead of one for each slab. Similarly, INFINISWAP block devices maintain one connection for each daemon instead of per-slab connections. Overall, for each active block device-daemon pair, there is one RDMA connection shared between the data plane and the control plane.

7 Evaluation

We evaluated INFINISWAP on a 32-machine, 56 Gbps Infiniband cluster on CloudLab [5] and highlight the results as follows:

- INFINISWAP provides 2×–4× higher I/O bandwidth than Mellanox nbdX [2]. While nbdX saturates 6 remote virtual cores, INFINISWAP uses none (§7.1).
- INFINISWAP improves throughputs of unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark by up to 4× (0.94×) to 15.4× (7.8×) over disk (nbdX) and tail latencies by up to 61× (2.3×) (§7.2).
- INFINISWAP ensures fast recovery from remote failures and evictions with little impact on applications; it does not impact remote applications either (§7.3).
- INFINISWAP benefits hold in a distributed setting; it increases cluster memory utilization by 1.47× using a small amount of network bandwidth (§7.4).

Experimental Setup. Unless otherwise specified, we use SlabSize = 1 GB, HeadRoom = 8 GB, HotSlab = 20 paging activities per second, and $\alpha = \beta = 0.2$ in all the experiments. For comparison, nbdX also utilizes remote memory for storing data.



(a) Bandwidth

(b) Remote CPU Usage

Figure 9: **INFINISWAP** provides higher read and write bandwidths without remote CPU usage, whereas Mellanox nbdX suffers from high CPU overheads and lower bandwidth.

Each of the 32 machines had 32 virtual cores and 64 GB of physical memory.

7.1 INFINISWAP Performance as a Block Device

Before focusing on INFINISWAP' effectiveness as a decentralized remote paging system, we focus on its raw performance as a block device. We compare it against nbdX and do not include disk because of its significantly lower performance. We used `fio` [6] – a well-known disk benchmarking tool – for these benchmarks.

For both INFINISWAP and nbdX, we performed parameter sweeps by varying the number of threads in `fio` from 1 to 32 and I/O depth from 2 to 64. Figure 9a shows the highest average bandwidth observed for different block sizes across all these parameter combinations for both block devices. In terms of bandwidth, INFINISWAP performs between 2× and 4× better than nbdX and saturates the 56 Gbps network at larger block sizes.

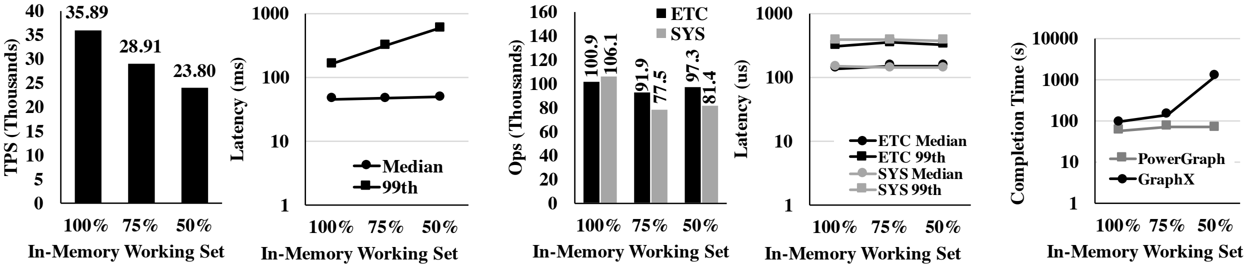
More importantly, we observe that due to nbdX's involvement in copying data to and from RAMdisk at the remote side, it has excessive CPU overheads (Figure 9b) and becomes CPU-bound for smaller block sizes. It often saturates the 6 virtual cores it runs on. In contrast, INFINISWAP bypasses remote CPU in the data plane and has close to zero CPU overheads in the remote machine.

7.2 INFINISWAP's Impact on Applications

In this section, we focus on INFINISWAP's performance on multiple memory-intensive applications with a variety of workloads (Figure 10) and compare it to that of disk (Figure 1) and nbdX (Figure 11).

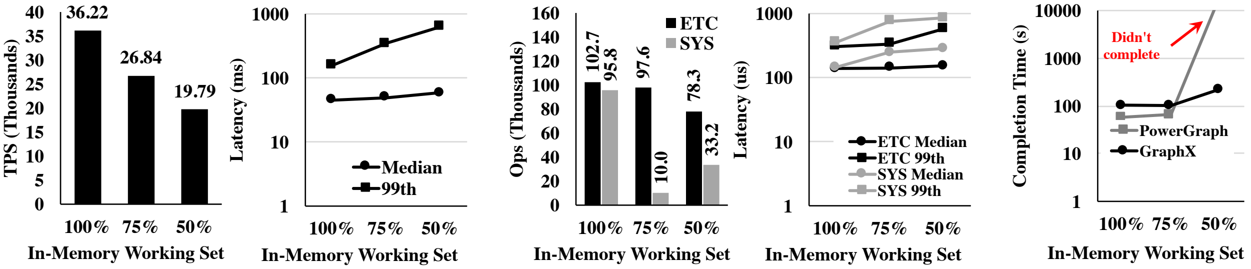
Workloads. We used four memory-intensive application and workload combinations:

1. TPC-C benchmark [22] on VoltDB [23];
2. Facebook workloads [26] on Memcached [18];
3. Twitter graph [52] on PowerGraph [45]; and
4. Twitter data on GraphX [46] and Apache Spark [77].



(a) TPC-C on VoltDB **(b)** Facebook workloads [26] on Memcached **(c)** Analytics

Figure 10: INFINISWAP performance for the same applications in Figure 1 for the same configurations. All plots show single-machine, server-side performance, and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite is true for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.



(a) TPC-C on VoltDB **(b)** Facebook workloads [26] on Memcached **(c)** Analytics

Figure 11: nbdX performance for comparison. All plots show single-machine, server-side performance, and the median value of five runs. Lower is better for the latency-related plots (lines) and the opposite is true for the throughout-related ones (bars). Note the logarithmic Y-axes in the latency/completion time plots.

Methodology. We focused on single-machine performance and considered three configurations – 100%, 75%, and 50% – for each application. We started with the 100% configuration by creating an `lxc` container with large enough memory to fit the entire workload in memory. We measured the peak memory usage, and then ran 75% and 50% configurations by creating containers with enough memory to fit those fractions of the peak usage. For INFINISWAP and nbdX, we use a single remote machine as the remote swap space.

7.2.1 VoltDB

VoltDB is an in-memory, transactional database that can import, operate on, and export large amounts of data at high speed, providing ACID reliability and scalability. We use its community version available on Github.

We use TPC-C to create transactional workloads on VoltDB. TPC-C performs 5 different types of transactions either executed on-line or queued for deferred execution to simulate an order-entry environment. We set 256 warehouses and 8 sites in VoltDB to achieve a reasonable single-container workload of 11.5 GB and run 2 million transactions.

We observe in Figure 10a that, unlike disk (Figure 1a), performance using INFINISWAP drops linearly instead of super-linearly when smaller amounts of workloads fit in memory. Using INFINISWAP, VoltDB experiences only a 1.5× reduction in throughput instead of 24× using disk in the 50% case. In particular, INFINISWAP improves VoltDB throughput by 15.4× and 99th-percentile latency by 19.7× in comparison to paging to disk. nbdX's performance is similar to that of INFINISWAP (Figure 11a).

Overheads of Paging. To understand why INFINISWAP's performance drops significantly when the workload does not fit into memory even though it is never paging to disk, we analyzed and compared its CPU and memory usage with all other considered applications (see Appendix A). We believe that because VoltDB is more CPU-intensive than most other memory-intensive workloads we considered, the overheads of paging (e.g., context switches) have a larger impact on its performance.

We note that paging-aware data structure placement (by modifying VoltDB) can help in mitigating this issue [36, 74]. We consider this a possible area of future work.

7.2.2 Memcached

Memcached is an in-memory object caching system that provides a simple key-value interface.

We use `memaslap`, a load generation and benchmarking tool for Memcached, to measure performance using recent data published by Facebook [26]. We pick ETC and SYS to explore the performance of INFINISWAP on workloads with different rates of SET operations. Our experiments start with an initial phase, where we use 10 million SET operations to populate a Memcached server. We then perform another set of 10 million queries in the second phase to simulate the behavior of a given workload. ETC has 5% SETs and 95% GETs. The key size is fixed at 16 bytes and 90% of the values are evenly distributed between 16 and 512 bytes [26]. The workload size is measured to be around 9 GB. SYS, on the other hand, is SET-heavy, with 25% SET and 75% GET operations. 40% of the keys have length from 16 to 20 bytes, and the rest range from 20 to 45 bytes. Values of size between 320 and 500 bytes take up 80% of the entire data, 8% of them are smaller, and 12% sit between 500 and 10000 bytes. The workload size is measured to be 14.5 GB. We set the memory limit in Memcached configurations to ensure that for 75% and 50% configurations it will respond using the swap space.

First, we observe in Figure 10b that, unlike disk (Figure 1b), performance using INFINISWAP remains steady instead of facing linear or super-linear drops when smaller amounts of workloads fit in memory. Using INFINISWAP, Memcached experiences only 1.03× (1.3×) reduction in throughput instead of 4× (17.4×) using disk for the 50% case for the GET-dominated ETC (SET-heavy SYS) workload. In particular, INFINISWAP improves Memcached throughput by 4.08× (15.1×) and 99th-percentile latency by 36.3× (61.4×) in comparison to paging to disk.

Second, nbdX does not perform as well as it does for VoltDB. Using nbdX, Memcached experiences 1.3× (3×) throughput reduction for the 50% case for the GET-dominated ETC (SET-heavy SYS) workload. INFINISWAP improves Memcached throughput by 1.24× (2.45×) and 99th-percentile latency by 1.8× (2.29×) in comparison to paging to nbdX. nbdX's performance is not very stable either (Figure 11b).

Pitfalls of Remote CPU Usage by nbdX. When the application itself is not CPU-intensive, the differences between INFINISWAP and nbdX designs become clearer. As paging activities increase (i.e., for the SYS workload), nbdX becomes CPU-bound in the remote machine; its performance drops and becomes unpredictable.

7.2.3 PowerGraph

PowerGraph is a framework for large-scale machine learning and graph computation. It provides parallel computation on large-scale natural graphs, which usually have highly skewed power-law degree distributions.

We run TunkRank [1], an algorithm to measure the influence of a Twitter user based on the number of that user's followers, on PowerGraph. TunkRank's implementation on PowerGraph was obtained from [9]. We use a Twitter dataset of 11 million vertices as the input. The dataset size is 1.3 GB. We use the asynchronous engine of PowerGraph and `tsv` input format with the number of CPU cores set to 2, resulting in a 9 GB workload.

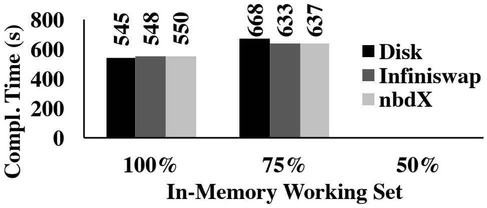


Figure 12: Comparative performance for PageRank using Apache Spark. The 50% configuration fails for all alternatives because Spark starts thrashing.

Figure 10c shows that, unlike disk (Figure 1c), performance using INFINISWAP remains stable. Using INFINISWAP, PowerGraph experiences only 1.24× higher completion time instead of 8× using disk in the 50% case. In particular, INFINISWAP improves PowerGraph's completion by 6.5× in comparison to paging to disk.

nbdX did not even complete at 50% (Figure 11c).

7.2.4 GraphX and Apache Spark

GraphX is a specialized graph processing system built on top of the Apache Spark in-memory analytics engine. We used Apache Spark 2.0.0 to run PageRank on the same Twitter user graph using both GraphX and vanilla Spark. For the same workload, GraphX could run using 12 GB maximum heap, but Spark needed 16 GB.

Figure 10c shows that INFINISWAP makes a 2× performance improvement over the case of paging to disk for the 50% configuration for GraphX. However, for Spark, all three of them fail to complete for the 50% configuration (Figure 12). In both cases, the underlying engine (i.e., Spark) starts thrashing – applications oscillate between paging out and paging in making little or no progress. In general, GraphX has smaller completion times than Spark for our workload.

7.3 Performance of INFINISWAP Components

So far we have considered INFINISWAP's performance without any eviction or failures. In this section, we analyze from both block device and daemon perspectives.

7.3.1 INFINISWAP Block Device

For these experiments, we present results for the 75% VoltDB configuration. We select VoltDB because it experienced one of the lowest performance benefits using INFINISWAP. We run it in one machine and distribute its slabs across 6 machines' remote memory. We then introduce different failure and eviction events to measure VoltDB's throughput loss (Figure 13).

Handling Remote Failures. First, we randomly turned off one of the 6 machines in 10 different runs; the failed (turned-off) machine did not join back. The average

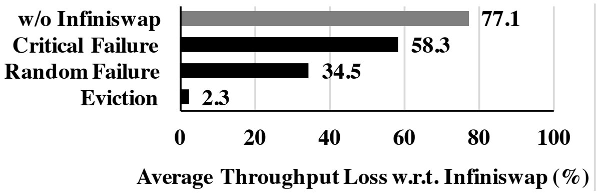


Figure 13: Average throughput loss of VoltDB with 75% in-memory working set w.r.t. INFINISWAP from different failure and eviction events. Lower is better.

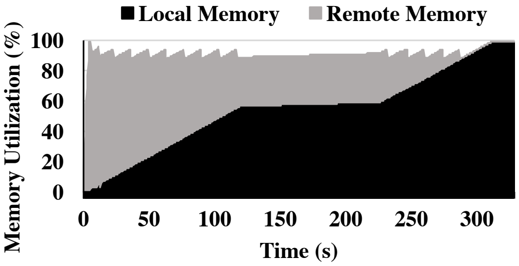


Figure 14: INFINISWAP daemon proactively evicts slabs to ensure that a local Memcached server runs smoothly. The white/empty region toward the top represents HeadRoom.

throughput loss was about 34.5% in comparison to INFINISWAP without any failures. However, we observed that the timing of failure has a large impact (e.g., during high paging activity or not). So, to create an adversarial scenario, we turned off one of the 6 machines again, but in this case, we failed the highest-activity machine during its peak activity period. The average throughout loss increased to 58.3%.

Handling Evictions. In this experiment, we evicted 1 slab from one of the remote machines every second and measured the average throughput loss to be about 2.3%, on average, for each eviction event (of 7–29 eviction-and-remapping events). As before, eviction of a high-activity slab had a slightly larger impact than that of one with lower activity.

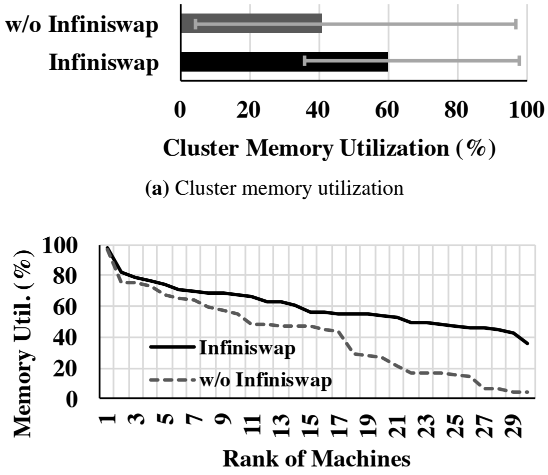
Time to Map a Slab. We also measured the time INFINISWAP takes to map (for the first time) or remap (due to eviction or failure) a slab. The median time was 54.25 milliseconds, out of which 53.99 went to Infiniband memory registration. Memory registration is essential and incurs the most interfering overhead in Infiniband communication [59]; it includes address translation, and pinning pages in memory to prevent swapping. Note that preallocation of slabs by INFINISWAP daemons mask close to 400 milliseconds, which would otherwise have been added on top of the 54.25 milliseconds. Detailed breakdown is given in Appendix B.2.

7.3.2 INFINISWAP Daemon

Now, we focus on INFINISWAP daemon's reaction time to increase in memory demands of local applications. For this experiment, we set HeadRoom to be 1 GB and started at time zero with INFINISWAP hosting a large number of remote slabs. We started a Memcached server soon after and started performing the ETC workload. Figure 14 shows how INFINISWAP daemon monitored local memory usage and proactively evicted remote slabs to make room – the white/empty region toward the top represents the HeadRoom distance INFINISWAP strived to maintain.

	ETC		SYS	
	W/o	With	W/o	With
Ops (Thousands)	95.9	94.1	96.0	93.5
Median Latency (us)	152.0	152.0	152.0	156.0
99**th Latency (us)**	319.0	318.0	327.0	343.0

Table 1: Performance of an in-memory Memcached server with and without INFINISWAP using remote memory.



(a) Cluster memory utilization

(b) Memory utilization of individual machines

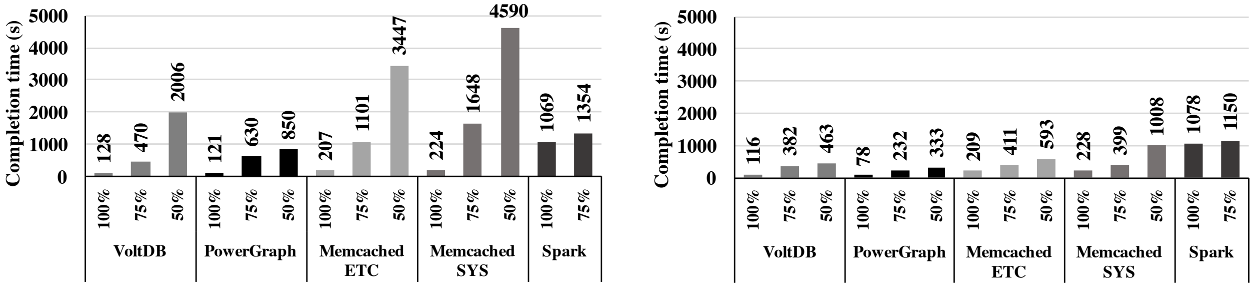
Figure 15: Using INFINISWAP, memory utilization increases and memory imbalance decreases significantly. Error bars in (a) show the maximum and the minimum across machines.

After about 120 seconds, when Memcached stopped allocating memory for a while, INFINISWAP stopped retreating as well. INFINISWAP resumed slab evictions when Memcached's allocation started growing again.

To understand whether INFINISWAP retreated fast enough not to have any impact on Memcached performance, we measured its throughput as well as median and 99th-percentile latencies (Table 1), observing less than 2% throughput loss and at most 4% increase in tail latency. Results for the SYS workload were similar.

Time to Evict a Slab The median time to evict a slab was 363 microseconds. A detailed breakdown of events is provided in Appendix B.2.

The eviction speed of INFINISWAP daemon can keep up with the rate of memory allocation in most cases. In extreme cases, the impact on application performance can be reduced by adjusting HeadRoom.



(a) Without Infiniswap

(b) Infiniswap

Figure 16: 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.

7.4 Cluster-Wide Performance

So far we have considered Infiniswap's performance for individual applications and analyzed its components. In this section, we deploy Infiniswap on a 32-machine RDMA cluster and observe whether these benefits hold in the presence of concurrency and at scale.

Methodology. For this experiment, we used the same applications, workloads, and configurations from Section 7.2 to create about 90 containers. We created an equal number of containers for each application-workload combination. About 50% of them were using the 100% configuration, close to 30% used the 75% configuration, and the rest used the 50% configuration.

We placed these containers randomly across 32 machines to create an memory imbalance scenario similar to those shown in Figure 2 and started all the containers at the same time. We measured completion times for the workload running each container; for VoltDB and Memcached completion time translates to transactions- or operations-per-second.

7.4.1 Cluster Utilization

Figure 15a shows that Infiniswap increased total cluster memory utilization by 1.47× by increasing it to 60% on average from 40.8%. Moreover, Infiniswap significantly decreased memory imbalance (Figure 15b): the maximum-to-median utilization ratio decreased from 2.36× to 1.6× and the maximum-to-minimum utilization ratio decreased from 22.5× to 2.7×.

Increase in Network Utilization. We also measured the total amount of network traffic over RDMA in the case of Infiniswap. This amounted to less than 1.88 TB over 1300 seconds across 32 machines or 380 Mbps on average for each machine, which is less than 1% of each machine's 56 Gbps interface.

7.4.2 Application-Level Performance

Finally, Figure 16 shows the overall performance of Infiniswap. We observe that Infiniswap's benefits are not restricted only to microbenchmarks, and it works well in the presence of cluster dynamics of many applications. Although improvements are sometimes lower than those observed in controlled microbenchmarks, Infiniswap still provides 3×–6× improvements for the 50% configurations.

8 Discussion and Future Work

Slab Size. For simplicity and efficiency, unlike some remote paging systems [38], Infiniswap uses moderately large slabs (SlabSize), instead of individual pages, for remote memory management. This reduces Infiniswap's meta-data management overhead. However, too large a SlabSize can lower flexibility and decrease space efficiency of remote memory. Selecting the optimal slab size to find a good balance between management overhead and memory efficiency is part of our future work.

Application-Aware Design. Although application transparency in Infiniswap provides many benefits, it limits Infiniswap's performance for certain applications. For example, database applications have hot and cold tables, and adapting to their memory access patterns can bring considerable performance benefits [74]. It may even be possible to automatically infer memory access patterns to gain significant performance benefits [36].

OS-Aware Design. Relying on swapping allows Infiniswap to provide remote memory access without OS modifications. However, swapping introduces unavoidable overheads, such as context switching. Furthermore, the amount of swapped data can vary significantly over time and across workloads even for the same application. Currently, Infiniswap cannot provide predictable performance without any controllable swap mechanism inside the OS. We would like to explore what can be done if we are allowed to modify OS-level decisions, such as changing its memory allocator or not making context switches due to swapping.

Application Differentiation. Currently, INFINISWAP provides remote memory to all the applications running on the machine. It cannot distinguish between pages from specific applications. Also, there are no limitations in remote memory usage for each application. Being able to differentiate the source of a page will allow us to manage resources better and isolate applications.

Network Bottleneck. INFINISWAP assumes that it does not have to compete with other applications for the RDMA network; i.e., the network is not a bottleneck. However, as the number of applications using RDMA increases, contentions will increase as well. Addressing this problem requires mechanisms to provide isolation in the network among competing applications.

9 Related Work

Resource Disaggregation. To decouple resource scaling and to increase datacenter efficiency, resource disaggregation and rack-scale computing have received significant attention in recent years, with memory disaggregation being the primary focus [10–12, 33, 49, 56, 57]. Recent feasibility studies [42, 53, 70] have shown that memory disaggregation may indeed be feasible even at a large scale, modulo RDMA deployment at datacenter-scale [48, 79]. INFINISWAP realizes this vision in practice and exposes the benefits of memory disaggregation to *any* user application without modifications.

Remote Memory Paging. Paging out to remote memory instead of local disks is a known idea [25, 31, 37, 39–41, 58, 64]. However, their performance and promises were often limited by slow networks and high CPU overheads. Moreover, they rely on central coordination for remote server selection, eviction, and load balancing. INFINISWAP focuses on a decentralized solution for the RDMA environment.

HPBD [55] and Mellanox nbdX [2] come the closest to INFINISWAP. Both of them can be considered as network-attached-storage (NAS) systems that use RAMdisk on the server side and are deployed over RDMA networks. However, there are several major differences that make INFINISWAP more efficient, resilient, and load balanced. First, they rely on remote RAMdisks, and data copies to and from RAMdisks become CPU-bound; in contrast, INFINISWAP does not involve remote CPUs, which increases efficiency. Second, they do not perform dynamic memory management, ignoring possibilities of evictions and subsequent issues. Finally, they do not consider fault tolerance nor do they attempt to minimize the impact of failures.

Software Distributed Shared Memory (DSM). DSM systems [29, 54, 65] expose a shared global address space to user applications. Traditionally, these systems have suffered from communication overheads to maintain coherence. To avoid coherence costs, the HPC community has favored the Partitioned Global Address Space (PGAS) model [30, 34] instead. However, PGAS systems require complete rewriting of user applications with explicit awareness of remote data accesses. With the advent of RDMA, there has been a renewed interest in DSM research, especially via the key-value interface [35, 51, 60, 63, 66, 69]. However, most of these solutions are either limited by their interface or require careful rethinking/rewriting of user applications. INFINISWAP, on the contrary, is a transparent, efficient, and scalable solution that opportunistically leverages remote memory.

10 Conclusion

This paper rethinks the well-known remote memory paging problem in the context of RDMA. We have presented INFINISWAP, a pragmatic solution for memory disaggregation without requiring any modifications to applications, OSes, or hardware. Because CPUs are not involved in INFINISWAP's data plane, we have proposed scalable, decentralized placement and eviction algorithms leveraging the power of many choices. Our in-depth evaluation of INFINISWAP on unmodified VoltDB, Memcached, PowerGraph, GraphX, and Apache Spark has demonstrated its advantages in substantially improving throughputs (up to $16.3\times$), median latencies (up to $5.5\times$), and tail latencies (up to $58\times$) over disks. It also provides benefits over existing RDMA-based remote memory paging solutions by avoiding remote CPU involvements. INFINISWAP increases the overall memory utilization of a cluster, and its benefits hold at scale.

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] A Twitter Analog to PageRank. http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.

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

[3] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2017-02-02.

[4] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.

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

[6] Fio - Flexible I/O Tester. https://github.com/axboe/fio.

[7] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2017-02-02.

[8] Google Container Engine. https://cloud.google.com/container-engine/.

[9] Graph Analytics Benchmark in CloudSuite. http://parsa.epfl.ch/cloudsuite/graph.html.

[10] HP Moonshot System: The world's first software-defined servers. http://h10032.www1.hp.com/ctg/Manual/c03728406.pdf.

[11] HP: The Machine. http://www.labs.hpe.com/research/themachine/.

[12] Intel RSA. http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html.

[13] Kubernetes. http://kubernetes.io.

[14] Linux memory management. http://www.tldp.org/LDP/tlk/mm/memory.html.

[15] Linux Multi-Queue Block IO Queueing Mechanism (blk-mq). https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq).

[16] Mellanox ConnectX-3 User Manual. http://www.mellanox.com/related-docs/user_manuals/ConnectX-3_VPI_Single_and_Dual_QSFP_Port_Adapter_Card_User_Manual.pdf.

[17] Mellanox SX6036G Specifications. http://www.mellanox.com/related-docs/prod_gateway_systems/PB_SX6036G.pdf.

[18] Memcached - A distributed memory object caching system. http://memcached.org.

[19] Microsoft Azure Cloud Services Pricing. https://azure.microsoft.com/en-us/pricing/details/cloud-services/. Accessed: 2017-02-02.

[20] Redis, an in-memory data structure store. http://redis.io.

[21] Stackbd: Stacking a block device. https://github.com/OrenKishon/stackbd.

[22] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[23] VoltDB. https://github.com/VoltDB/voltdb.

[24] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[25] E. A. Anderson and J. M. Neefe. An exploration of network RAM. Technical Report UCB/CSD-98-1000, EECS Department, University of California, Berkeley, Dec 1994.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[27] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[28] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[29] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP*, 1991.

[30] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[31] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[32] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[33] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A network stack for rack-scale computers. In *SIGCOMM*, 2015.

[34] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing*, 1993.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[36] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.

[37] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[38] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[39] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 201–212. ACM, 1995.

[40] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar 1991.

[41] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Journal of Cluster Computing*, 2(4):281–293, 1999.

[42] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[43] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[44] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[46] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[47] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[48] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.

[49] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[50] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[51] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[52] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[53] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD*, 2016.

[54] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, 1989.

[55] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[56] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[57] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[58] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX ATC*, 1996.

[59] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the memory registration process in the Mellanox Infiniband software stack. In *Euro-Par*, 2006.

[60] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*, 2013.

[61] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[62] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[63] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[64] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[65] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[66] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[67] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[68] G. Park. Brief announcement: A generalization of multiple choice balls-into-bins. In *PODC*, 2011.

[69] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[70] P. S. Rao and G. Porter. Is memory disaggregation feasible?: A case study with Spark SQL. In *ANCS*, 2016.

[71] C. Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[72] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[73] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[74] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *Ninth International Workshop on Data Management on New Hardware*, 2013.

[75] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[76] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

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

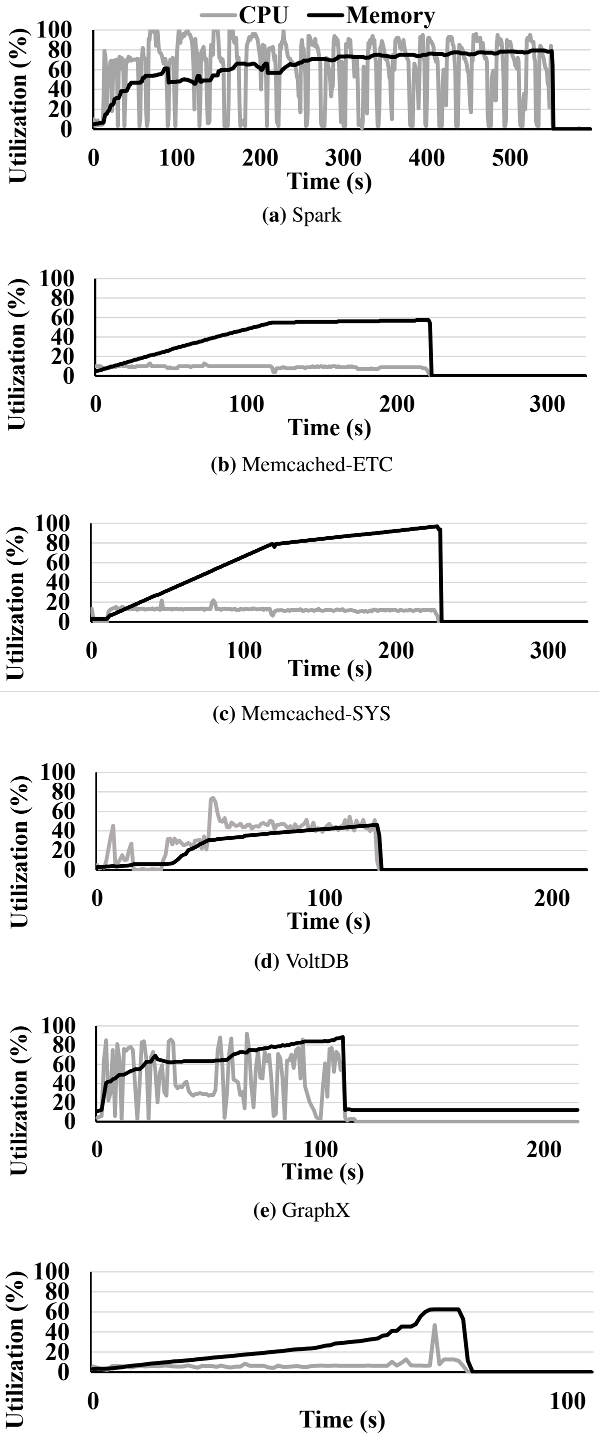
[78] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[79] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.

A Characteristics of the Benchmarks

We ran each benchmark application in separate containers with 16 GB memory (32 GB only for Spark) and measured their real-time CPU and memory utilizations from cold start. We make the following observations from these experiments.

First, while memory utilizations of all applications increased gradually before plateauing, Spark has significantly higher memory utilization along with very high CPU usage (Figure 17a). This is perhaps one of the primary reasons why Spark starts thrashing when it cannot keep its working set in memory (i.e., in 75% and 50% configurations). While GraphX exhibits a similar trend (Figure 17e), its completion time is much smaller for the same workload. Even though it starts thrashing in the



(a) Spark

(b) Memcached-ETC

(c) Memcached-SYS

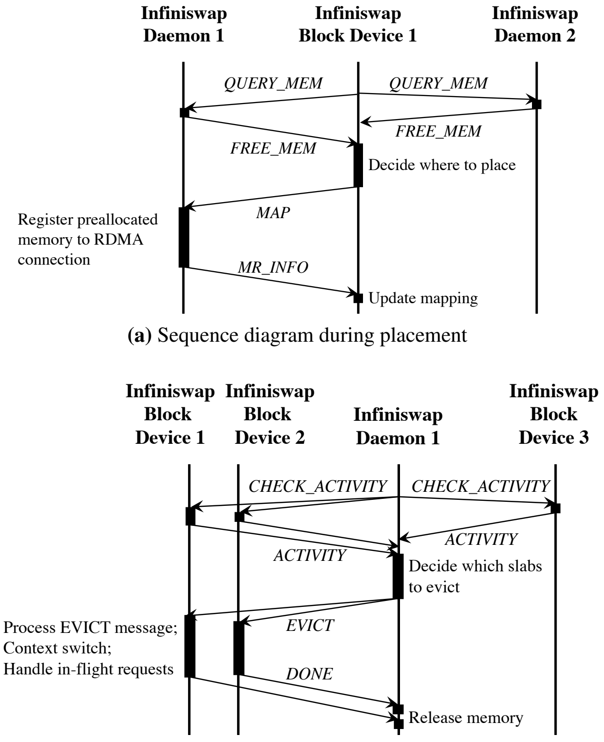
(d) VoltDB

(e) GraphX

(f) PowerGraph

Figure 17: CPU and memory usage characteristics of the benchmarked applications and workloads running on containers with 16 GB memory (32 GB only for Spark). Note the increasingly smaller timescales in different X-axes due to smaller completion times of each workload.

50% configuration, it can eventually complete before spiraling out of control.


Infiniswap Daemon 1
Infiniswap Block Device 1
Infiniswap Daemon 2
QUERY_MEM
QUERY_MEM
FREE_MEM
FREE_MEM
Decide where to place
Register preallocated memory to RDMA connection
MAP
MR_INFO
Update mapping
Infiniswap Block Device 1
Infiniswap Block Device 2
Infiniswap Daemon 1
Infiniswap Block Device 3
CHECK_ACTIVITY
CHECK_ACTIVITY
ACTIVITY
ACTIVITY
Decide which slabs to evict
Process EVICT message; Context switch; Handle in-flight requests
EVICT
DONE
Release memory

(a) Sequence diagram during placement

(b) Sequence diagram during eviction

Figure 18: Decentralized placement and eviction in INFINISWAP. (a) INFINISWAP block devices use the power of two choices to select machines with the most available memory to place each slab. (b) INFINISWAP daemons use the power of many choices to select slab(s) to evict; in this case, the daemon is contacting 3 block devices to evict 2 slabs.

Second, other than Spark and GraphX, VoltDB has at least 3× higher average CPU utilization than other benchmarks (Figure 17d). This is one posible explanation of its smaller improvements with INFINISWAP for the 50% and 75% cases in comparison to other less CPU-heavy applications – overheads of paging (e.g., context switch) was possibly a considerable fraction of VoltDB's runtimes.

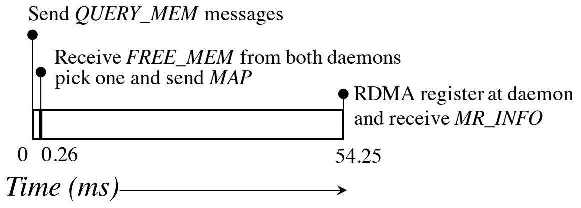
Third, both ETC and SYS workloads gradually allocate more memory over time, but ETC plateaus early because it has mostly GET operations (Figure 17b), whereas SYS keeps increasing because of its large number of SET operations (Figure 17c).

Finally, PowerGraph is the most efficient of the workloads we considered (Figure 17f). It completes faster and has the smallest resource usage footprint, both of which contribute to its consistent performances using INFINISWAP across all configurations.

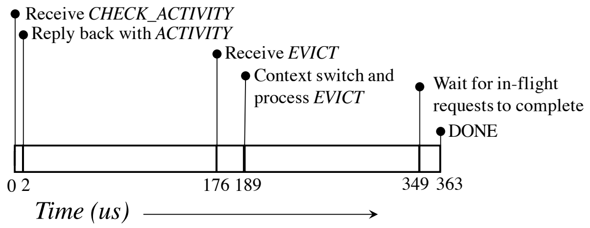
B Control Plane Details

B.1 Control Messages

INFINISWAP components use message passing to transfer memory information and memory service agreements.


Send *QUERY_MEM* messages
Receive *FREE_MEM* from both daemons pick one and send *MAP*
RDMA register at daemon and receive *MR_INFO*
0
0.26
54.25
Time (ms)

(a) Timing diagram of placement


Receive *CHECK_ACTIVITY*
Reply back with *ACTIVITY*
Receive *EVICT*
Context switch and process *EVICT*
Wait for in-flight requests to complete
DONE
0
2
176
189
349
363
Time (us)

(b) Timing diagram of eviction

Figure 19: Timing diagrams (not drawn to scale) from a INFINISWAP block device's perspective during decentralized placement and eviction events.

There are eight message types; the first four of them are used by the placement algorithm (Figure 18a) and the rest are used by the eviction algorithm (Figure 18b).

1. *QUERY_MEM*: Block devices send it to get the number of available memory slabs on the daemon.
2. *FREE_MEM*: Daemons respond to *QUERY_MEM* requests with the number of available memory slabs.
3. *MAP*: Block device confirms that it has decided to use one memory slab from this daemon.
4. *MR_INFO*: Daemon sends memory registration information (`rkey`, `addr`, `len`) of an available memory slab to the block device in response to MAP.
5. *CHECK_ACTIVITY*: Daemons use this message to ask for paging activities of specific memory slab(s).
6. *ACTIVITY*: Block device's response to the *CHECK_ACTIVITY* messages.
7. *EVICT*: Daemons alert the block device which memory slab(s) it has selected to evict.
8. *DONE*: After completely redirecting the requests to the to-be-evicted memory slab(s), block device responds with this message so that the daemon can safely evict and return physical memory to its local OS.

B.2 Breakdown of Control Plane Timings

Here we breakdown the timing results from Section 7.3. Figures 19a and 19b provide details of placement and eviction timings from a INFINISWAP block device's perspective.

Parameter	Value
Datacenter OPEX	$0.04/W/month
Electricity Cost	$0.067/kWh
InfiniBand NIC Power	8.41W [16]
InfiniBand Switch Power	231W [17]
Power Usage Effectiveness (PUE)	1.1

Table 2: Cost Model Parameters [27].

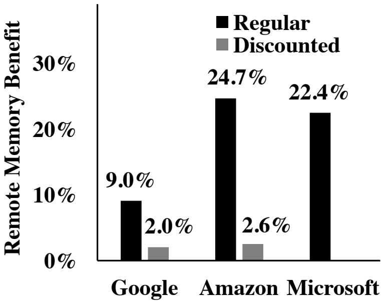


Figure 20: Revenue increases with INFINISWAP under three different cloud vendors' regular and discounted pricing models.

C Cost-Benefit Analysis

In many production clusters, memory and CPU usages are unevenly distributed across machines (§2.3) and resources are often underutilized [72, 78]. Using memory disaggregation via INFINISWAP, machines with high memory demands can use idle memory from other machines, thus enabling more applications to run simultaneously on a cluster and providing more economic benefits. Here we perform a simple cost-benefit analysis to get a better understanding of such benefits.

We limit our analysis only to RDMA-enabled clusters, and therefore, do not consider capital expenditure and depreciation cost of acquiring RDMA hardware. The major source of operational expenditure (OPEX) comes from the energy consumption of Infiniband devices – the parameters of our cost model are listed in Table 2. The average cost of INFINISWAP for a single machine is around $1.45 per month.

We also assume that there are more idle CPUs than idle memory in the cluster, and INFINISWAP's benefits are limited by the latter. For example, on average, about 40% and 30% of allocated CPUs and memory are reported to remain unused in Google clusters [72]. We use the price lists from Google Cloud Compute [7], Amazon EC2 [3], and Microsoft Azure [19] to build the benefit model. INFINISWAP is found to increase total cluster memory utilization by around 20% (7.4.1), varying slightly across different application deployment scenarios. We assume that there are 20% physical memory on each machine that has been allocated to local applications but the remainder is used as remote memory via INFINISWAP. The additional benefit is then the price of

20% physical memory after deducting the cost of operating Infiniband.

There are several price models from different vendors. In a model we call the *regular pricing model*, resource availability is strictly guaranteed. In another model from Google (preemptible instance) and Amazon (spot instance), resource can be preempted or become unavailable based on resource availability in a cluster. Of course, the resource price in the latter model is much lower than the regular model. We call it the *discounted pricing model*.

If INFINISWAP can ensure unnoticeable performance degradation to applications, remote memory can be counted under regular pricing; otherwise, discounted pricing should be used. Figure 20 shows benefits of INFINISWAP. With an ideal INFINISWAP, cluster vendors can gain up to 24.7% additional revenue. If we apply the discounted model, then it decreases to around 2%.