# Accelerating Disaggregated Data Centers Using Unikernel

Wonsup Yoon
KAIST
wsyoon@kaist.ac.kr

Jinyoung Oh
KAIST
jinyoungoh@kaist.ac.kr

Sue Moon
KAIST
sbmoon@kaist.edu

Youngjin Kwon
KAIST
yjkwon@kaist.ac.kr

## CCS CONCEPTS

• **Networks** → **Cloud computing**.

## KEYWORDS

Disaggregated data center, Unikernel, RDMA

**ACM Reference Format:**
Wonsup Yoon, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2020. Accelerating Disaggregated Data Centers Using Unikernel. In *ACM Special Interest Group on Data Communication (SIGCOMM '20 Demos and Posters), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3405837.3411397

## 1 INTRODUCTION

Resource disaggregation is a new hardware and system paradigm to split computation, memory, and storage into individual resource pools. Compared with the traditional single machine design, resource disaggregation allows independent scaling and managed grouping of hardware resources beyond the boundary of a single machine. It is a key driver to solve the low resource utilization [2] and to hide intermittent hardware failures [13, 14] in cloud systems. Cloud service providers are building disaggregated data centers (DDC) to take full advantage of resource disaggregation [8].

Memory disaggregation puts CPUs and memory in physically separate nodes. CPU nodes have small local memory for caching pages from remote memory nodes. Increased complexity of disaggregated resource management raises questions such as: "How should we fetch and evict remote pages in order to utilize local memory efficiently?" and "How should we prefetch in order to hide network latency?" There are a myriad of cloud applications and they have different requirements for the questions.

Unikernels are library operating systems where the kernel and an application run in a single address space under privileged mode (ring-0) [5–7, 9, 10]. Unikernels bring many advantages to memory disaggregation: remote paging without mode switching and ease of application-specific customization. Remote pagingis a widely used disaggregated memory management technique that swaps in remote pages and out local pages using a memory management unit (MMU) [1, 8, 12]. Because accessing an MMU requires privileged mode, traditional OSes must switch modes via a slow system call. In unikernels, on the other hand, applications access the MMU directly and avoid overhead from mode switching. Another advantage of unikernels is ease of customization for its application.

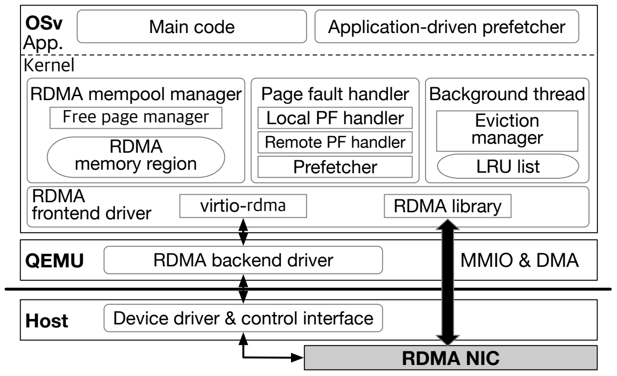

**Figure 1: System overview**

Because a unikernel serves only one application, developers are able to customize the unikernel without concerns over side effects to other applications. Applications freely set their own resource management policies. Moreover, unikernels allow applications to drive their own policies with full knowledge of their memory usage patterns. Our goal of this work is to optimize remote paging using unikernels in DDCs.

*Unikernel-based memory disaggregation.* We built a disaggregated memory system in an actively developing unikernel, OSv [7]. Figure 1 shows a system overview of modules in a compute node. An OSv kernel, linked with an application, runs on QEMU. It interacts with host Linux which runs as a hypervisor. We use RDMA over Infiniband as an efficient data path between a compute node and a memory node. Compared to TCP/IP, RDMA has a hardware support for fetching contiguous memory larger than MTU. This enables an efficient exchange of pages between nodes. We add RDMA frontend and backend drivers to the system to use RDMA. The data path (thick arrow) bypasses complex kernel and virtualization layers, directly communicating with an RDMA NIC. Hypervisor exposes a control path (thin arrow) via virtualized interfaces, and the interfaces are redirected to applications through our unikernel. We redesign the OSv's memory system to evict and fetch pages between a compute node and a memory node. We add an RDMA mempool manager to manage local pages, a page fault handler to swap in remote pages, and a background thread to swap out local pages.

*Application-driven prefetching.* A unikernel has only one application, which lies in the same address space, and communicates with it via low overhead function calls. It allows the kernel to ask its application to drive kernel's functionalities. In this work we demonstrate an application-driven prefetcher as an example scenario. The application-driven prefetcher uses the application's domain knowledge to establish an efficient prefetching policy. Our OSv unikernel interacts with the prefetcher via upcalls. When a page fault happens, OSv's page fault handler calls the application-level prefetcher. The application-level prefetcher determines which pages to access and informs the page fault handler about the decision to fetch those pages.

## 2 CASE STUDY: REDIS

To show the effectiveness of our approach, we conduct a case study of Redis, a popular in-memory data structure store [11]. In applications such as Redis, the memory access pattern is highly unpredictable, for it is only known upon request arrivals. Below we demonstrate the performance improvement our application-driven prefetcher delivers in Redis workloads. The prefetcher decides which pages to retrieve using hints from Redis's data structures.

*Accelerating a prefetcher for Redis in a unikernel.* Redis uses the Simple Dynamic Strings (SDS) library to store keys and values. Redis's SDS consists of a header and data followed by a terminal character. The prefetcher uses the length information in the header to decide the number of pages to retrieve.

Another use case of the application-driven policy is prefetching Redis objects from range queries. Redis supports an `LRANGE` operation which returns a set of objects from a list specified by the range operator. For example, `LRANGE my_list 0 100` returns objects from index 0 to 100 in `my_list`. The range parameter is a good hint for the number of pages to prefetch in order to serve the range query.

We do not modify the Redis main code but only link 155 LoC of our application-level prefetcher.

*Evaluation.* To evaluate the benefits of our unikernel-based memory disaggregation and application-driven prefetcher, we compare our prototypes without any prefetcher (*OSv w/o prefetcher*), with Linux-like VMA-based sequential prefetcher [4] (*OSv w/ sequential prefetcher*), and with an application-driven prefetcher (*OSv w/ app-driven prefetcher*) to the state-of-the-art disaggregated memory system, Infiniswap [1]. Figure 2 shows the performance improvements of our unikernel-based disaggregated memory system with an application-driven prefetcher under `GET` workload and `LRANGE` workload. The `GET` workload has six equally distributed data sizes —4KB, 8KB, 16KB, 32KB, 64KB, and 128KB—that represent data sizes of more than 80% of objects in the Facebook's photo servers [3], and the `LRANGE` workload consists of range queries over 100 thousand distinct lists that have 200 elements in average. Both workloads have a working set of about 20GB big. We evaluate the performance of Redis with 100% (`20GB+`), 50% (`10GB`), 25% (`5GB`), and 12.5% (`2.5GB`) local memory of the working set.

The result shows that two notable points. First, *OSv w/o prefetcher* outperforms *Infiniswap* under all workloads and memory configuration (up to 2.1× higher throughput and up to 4.9× better 99th tail latency). It is mainly due to the fast, simple IO path of unikernel and our efficient implementation of fetching remote pages. Second, *OSv w/ app-driven prefetcher* outperforms *OSv w/ sequential prefetcher* especially under the `LRANGE` workload. Under simple `GET` workloads, there is little difference among prefetchers. Given complex `LRANGE` memory accesses, on the other hand, our application-driven prefetcher outperforms the sequential prefetcher (up to 1.7×). The

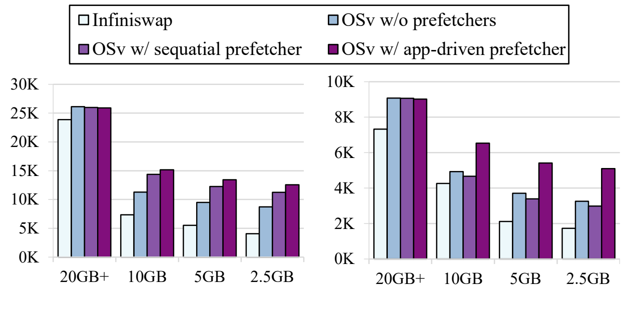

**Figure 2: Redis's request handling throughput (requests per second) under `GET` and `LRANGE` workloads with `20GB+`, `10GB`, `5GB`, `2.5GB` local memory. Each workload has about 20GB of working set. Higher is better.**

application-driven prefetcher uses its application knowledge to improve performance, while the sequential prefetcher degrades performance with wrong decisions. Overall, *OSv w/ app-driven prefetcher* with 2.5GB local memory is up to 75% faster than *Infiniswap* with even 10GB local memory. Our system is more effective when its local memory is highly limited (`2.5GB`). By adding only 155 LoC of a prefetcher, it shows a 2.9× higher throughput compared to Infiniswap.

## ACKNOWLEDGEMENT

This research was supported by Energy Cloud R&D Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2019M3F2A1072211).

## REFERENCES

[1] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. USENIX Association.
[2] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *IEEE/ACM International Workshop on Quality of Service (IWQoS '19)*. ACM.
[3] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An Analysis of Facebook Photo Caching. In *SOSP (SOSP '13)*. ACM.
[4] Ying Huang. 2017. mm, swap: VMA based swap readahead. https://lwn.net/Articles/716296/.
[5] IncludeOS. 2020. IncludeOS - Run your application with zero overhead. https://www.includeos.org.
[6] Rump kernel community. 2020. Rump Kernels. http://rumpkernel.org.
[7] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *USENIX Annual Technical Conference (ATC '16)*. USENIX Association.
[8] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. ACM.
[9] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM.
[10] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels:

The Next Stage of Linux's Dominance. In *Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM.

[11] Salvatore Sanfilippo. 2020. Redis. https://redis.io.

[12] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association.

[13] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing Cloud Computing Hardware Reliability. In *ACM Symposium on Cloud Computing (SoCC '10)*. ACM.

[14] G. Wang, L. Zhang, and W. Xu. 2017. What Can We Learn from Four Years of Data Center Hardware Failures?. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '17)*. IEEE.