Rethinking Software Runtimes for Disaggregated Memory

Irina Calciu
VMware Research
USA

M. Talha Imran
Penn State University
USA

Ivan Puddu
ETH Zürich
Switzerland

Sanidhya Kashyap
EPFL
Switzerland

Hasan Al Maruf
University of Michigan
USA

Onur Mutlu
ETH Zürich
Switzerland

Aasheesh Kolli
Penn State University/Google
USA

ABSTRACT

Disaggregated memory can address resource provisioning inefficiencies in current datacenters. Multiple software runtimes for disaggregated memory have been proposed in an attempt to make disaggregated memory practical. These systems rely on the virtual memory subsystem to transparently offer disaggregated memory to applications using a local memory abstraction. Unfortunately, using virtual memory for disaggregation has multiple limitations, including high overhead that comes from the use of page faults to identify what data to fetch and cache locally, and high dirty data amplification that comes from the use of page-granularity for tracking changes to the cached data (4KB or higher).

In this paper, we propose a fundamentally new approach to designing software runtimes for disaggregated memory that addresses these limitations. *Our main observation is that we can use cache coherence instead of virtual memory for tracking applications' memory accesses transparently, at cache-line granularity.* This simple idea (1) eliminates page faults from the application critical path when accessing remote data, and (2) decouples the application memory access tracking from the virtual memory page size, enabling cache-line granularity dirty data tracking and eviction. Using this observation, we implemented a new software runtime for disaggregated memory that improves average memory access time by 1.7-5X and reduces dirty data amplification by 2-10X, compared to state-of-the-art systems.

CCS CONCEPTS

• **Software and its engineering** → **Distributed memory**.

KEYWORDS

disaggregated memory, remote memory, cache coherence.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446713

ACM Reference Format:
Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446713

1 INTRODUCTION

To meet modern applications' stringent low latency and high throughput demands, datacenter administrators over-provision monolithic servers to account for peak demand. As a consequence, datacenter memory utilization is low, stagnating around 65% [78]. In addition, applications' ever increasing demands for memory often translate into the need to frequently upgrade the computing infrastructure altogether, as the monolithic server model is based on a tight coupling between computing resources and memory. This tight coupling is also problematic in the face of hardware failures, since the entire server will stop working when one component fails. All these inefficiencies add up and increase datacenter costs [18].

Disaggregated memory addresses these problems by improving memory utilization and decreasing the total memory overprovisioning necessary to avoid out-of-memory errors or swapping [11]. In addition, disaggregated memory enables independent scaling of memory and compute, and it disentangles hardware failures and replacements from the monolithic server. Fine-grain microsecond-latency networking technologies, such as Remote Direct Memory Access (RDMA) and Gen-Z [33], make hardware disaggregated memory feasible in the near future [37, 79, 82].

Unfortunately, enabling applications to efficiently adopt disaggregated memory is not straightforward. Software runtimes [10, 15, 36, 57, 72] have been proposed to enable applications to *transparently*, without code changes, use remote memory – the memory of another host in the rack or memory that has been physically disaggregated from the compute. These systems use various kernel subsystems [10, 36, 72] or redesign the kernel altogether [71]. Fundamentally, they all rely on the core virtual memory mechanism for three essential functions: (1) *fetching and caching remote data* by first detecting remote accesses using *page faults*, then caching the remote pages in a local DRAM cache; (2) *tracking dirty data* among the cached pages by write-protecting the pages and causing a *write page fault* on the first write to each page; and (3) *evicting*

cached pages from the local DRAM cache, which requires marking the pages as not present and flushing the translation look-aside buffers (TLBs).

Virtual memory provides application transparency, but results in high overhead and causes a significant drop in application performance, even when the amount of remote data accessed is small (§2). Page faults incur high latencies, exceeding network latencies, which makes the software stack a bottleneck for accessing remote memory. Moreover, virtual memory requires moving and tracking data at page-granularity, with a page size of 4KB or higher. In contrast, throughout their lifetimes, applications write a small part of each page, typically only 1-8 cache-lines out of 64, causing large dirty data amplification and poor network utilization, by re-writing the same data that is already in remote memory. We analyzed multiple production-quality applications and measured a dirty data amplification between 2X and 31X for 4KB pages (§2).

Some systems [29, 30, 61] avoid page faults and work at a finer-granularity than pages (objects), but require specialized application changes and thus sacrifice transparency. In practice, rewriting existing applications for remote memory is error-prone and requires expensive engineering resources and expertise. Thus, our focus is on application-transparent mechanisms that do not require any source code changes or recompilation.

Overall, there is a mismatch between applications' requirements for remote memory—low latency, fine granularity, application-transparency—and the virtual memory mechanism, which has not been designed to provide low-latency or fine-granularity. **Our key insight is that cache coherence can provide better hardware primitives to support disaggregated memory, by transparently tracking applications' memory accesses at cache-line granularity, without page faults.**[1] We describe a reference architecture that provides the necessary primitives to software using future cache-coherent field programmable gate arrays (§2.3). We expect such hardware support to become available in the near future, with the adoption of CXL-based platforms [74].

In this paper, we propose a principled approach to building software runtimes for remote memory based on the new hardware primitives (§3). We designed and implemented Kona, a software runtime that rethinks the design of each of the three remote memory functions performed by virtual memory in current systems (fetching remote memory, tracking dirty data and evicting cached pages) to rely on new hardware primitives enabled by the cache coherence protocol (§4). Kona moves high-overhead virtual memory operations off the critical path of execution, and tracks dirty cache-lines, decoupling tracking and movement from the virtual memory page size, for a 6.6X speedup.

The challenge is to accurately evaluate Kona's benefits and overheads using current architectures, despite the lack of hardware support. To solve this challenge, we developed several tools (§5) that allow us to simulate or emulate the necessary hardware primitives. We used them to measure Kona's benefits for the three types of remote memory operations: (1) Kona improves fetching remote memory by eliminating page faults, resulting in a reduced average memory access time by 1.7X and 5X compared to LegoOS [71] and Infiniswap [71], respectively. (2) Kona improves performance of dirty data tracking by 35% compared to page granularity write-protection, while reducing write amplification by 2-10X, by using cache-line granularity. (3) Kona improves eviction efficiency by writing only modified cache-lines back to the disaggregated memory, which improves network goodput by 4-5X.

In summary, we make the following contributions:

- We analyze the shortcomings of current runtimes for remote memory and show that they result in large overhead and dirty data amplification (§2).
- We propose a new approach for disaggregated memory software, using hardware primitives for remote memory caching and cache-line dirty data tracking based on cache coherence; we describe a reference architecture that can implement these primitives (§4.3).
- We design and implement Kona, a software runtime that can use the new hardware primitives for efficient execution (§4).
- We design and implement multiple emulation and simulation tools (§5) and we use them to measure Kona's solutions for the three types of remote memory operations (§6).

The code is available at github.com/project-kona/asplos21-ae [24].

2 BACKGROUND AND MOTIVATION

Object-based remote memory systems can provide fine-grain access to remote data and expose a key-value or a data-structure-based interface [29, 30, 61]. These systems achieve good performance, but rely on application semantic information to allocate or access data in remote memory and require intrusive code changes to port legacy applications. In contrast, *page-based remote memory systems* rely on OS-level mechanisms and interfaces, such as swapping or file systems, to offer remote memory to applications (almost[2]) transparently [10, 36, 45, 72]. However, the ease of programmability comes at the cost of forcing coarse-grain (page-granularity) access to remote memory through expensive OS code paths, leading to performance degradation and memory overhead.

Page-based and object-based systems achieve different tradeoffs in the remote memory space. We seek to leverage the benefits from both (Table 1). Our goal is to support legacy applications without changing them. Our system, Kona, is most similar to page-based remote memory,[3] but it leverages cache coherence mechanisms to achieve cache-line granularity dirty data tracking and to avoid page faults on the application critical path. Unlike current systems, Kona requires hardware support.

Table 1: Taxonomy of remote memory systems.

Remote memory	Granularity	Programability	Mechanism
Page-based [10, 36]	coarse (page) (-)	*transparent (+)*	virtual memory
Object-based [29, 30, 61]	*fine (object) (+)*	app-specific (-)	code changes
Kona	*cache-line (+)*	*transparent (+)*	cache coherence

Page-based systems work by mapping remote memory in an application's address space and using virtual memory mechanisms to cache remote pages into a local software-managed DRAM cache.

[1] We extend a workshop paper [25] by implementing a runtime system and multiple tools for simulation and emulation.

[2] Page-based systems can be completely transparent to the application, or offer additional features using small application modifications.

[3] Thus, in this paper we compare only to page-based remote memory systems.

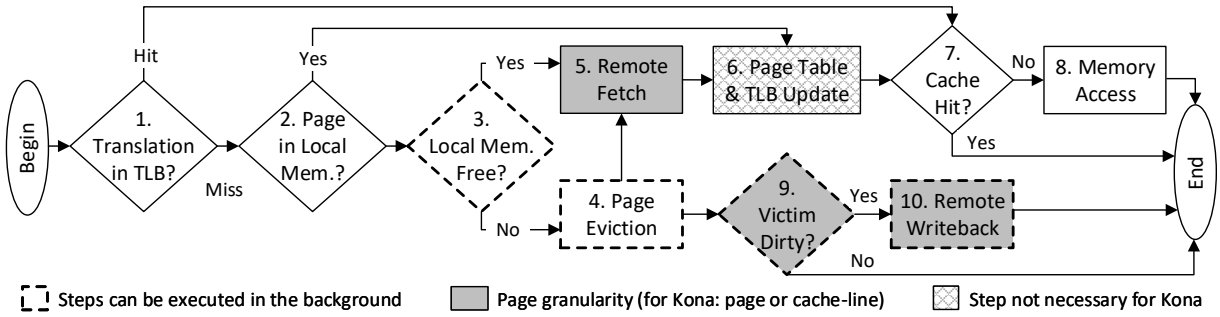


Figure 1: Life of a memory access in a remote memory system

Fig. 1 shows the different steps in the life of a memory access in a page-based remote memory system, which has to support three main operations:

1. Fetching remote data. When an application attempts to read or write data that is not present in the local DRAM cache, a page fault is triggered and the page is fetched from the remote host ⑤ by a custom page fault handler. Once the page has been fetched, the page tables and TLBs are appropriately modified ⑥ and the memory access moves forward. However, in certain scenarios, there might not be enough free space in the local DRAM cache to insert a new page, causing an eviction mechanism to make room for incoming pages ④.

2. Tracking dirty data. Locally cached pages that have been modified need to be written back to remote hosts to avoid data loss. Keeping track of modified pages is called *dirty data tracking* and is realized using *write page faults*: a page is initially marked as read-only when it is first fetched into the local DRAM cache and when the application tries to modify it, a page fault will be triggered to disable the write-protection on the page. The page is marked dirty when the write page fault is serviced. If an eviction candidate page is dirty ⑨ then a writeback operation is issued to the remote host ⑩.

3. Evicting local data back to remote memory. Periodically, the local cache evicts some cached remote pages to make room for new remote pages. Pages chosen for eviction that have not been modified since they were last brought into the local DRAM cache can be silently evicted.

2.1 Current Remote Memory Shortcomings

All three remote memory operations above suffer from high overhead, because they rely on virtual memory mechanisms. We describe their shortcomings next.

High overhead in fetching remote data. Page-based remote memory systems incur large overheads due to page faults and TLB invalidations. We ran Redis [7], a data-structure server application, with a state-of-the-art remote memory system (Infiniswap) and we observed that moving as little as 25% of the application's data remotely causes the throughput to drop by more than 60%. The explanation for this huge degradation is given by the high remote data access latency: we measured Infiniswap's remote access latency to be over 40μs. LegoOS, a new operating system designed for disaggregated memory, incurs a 10μs latency for a remote fetch operation. This high latency is astonishing, considering that a 4KB RDMA read operation is generally as fast as 3μs. The difference is all caused by the software stack and is not specific to Infiniswap or LegoOS. Other page-based remote memory frameworks also experience similar degradation [10, 72] due to their reliance on page faults to fetch remote data [11, 25, 32, 50]. Moreover, page faults cause the processor to flush its instruction pipeline, pollute CPU caches, and reduce the CPU prefetcher's effectiveness, as it cannot prefetch past a page fault. Unfortunately, there is no silver bullet to improve the latency. It is the compounded result of a sum of small operations, such as finding and allocating virtual memory areas (VMAs), managing the page cache and the LRU list [57]. Our approach is to move these operations off the critical path of the application.

Overhead in dirty data tracking and eviction. We measured a 35% decrease in throughput for Redis due to write page faults (§6). In addition, write-protecting pages requires modifying page tables and invalidating the TLBs, during which time the application is not running. The time an application is stopped increases with the size of its memory. The overheads are even higher for large pages, which first get broken down to 4KB pages to decrease the amplification [77]. Evicting pages also requires changing their protection, which incurs additional TLB invalidations on top of the ones required for dirty data tracking. We measured that eviction latencies could be over 32 μs with Infiniswap even though an RDMA 4KB write takes 3μs. Similarly to the remote fetch latency, we cannot expect to improve performance significantly by optimizing a small number of functions. The eviction and dirty tracking overhead adds up from many small operations: checking if the page is locked, checking for other PTEs referencing the page, unmapping the page, clearing the dirty bit, flushing the TLB, etc.

High dirty data amplification. We define amplification as the ratio of data marked as dirty using the tracking granularity to the actual number of bytes written by the application. For example, if the application writes 1KB of data within a page, with 4KB-tracking the entire page is marked dirty, so the dirty data amplification is 4. Often, applications access only a small part of a page [9]. Therefore, using page granularity for tracking dirty data results in high amplification and poor network utilization, because more data is transferred over the network than necessary. We used dynamic binary instrumentation with the Intel Pin tool [5] to measure dirty data amplification in real applications. We split each application's

execution to discrete time windows (10 seconds) and measure the behavior online in each window. We consider the following applications: 1) Redis [7], a data structure server, running uniformly random (Rand) and sequential (Seq) workloads; 2) GraphLab [52] - a graph analytics framework, running Page Rank, Graph Coloring, Connected Components and Label Propagation; 3) Metis [56], an in-memory MapReduce framework running Linear Regression and Histogram and 4) VoltDB [8], an in-memory column store database, running a TPC-C workload. Memory allocated and used by these applications varies between 133MB (Redis Seq) and 40GB (Linear regression and Histogram). We measured dirty data amplification for different tracking granularities: 4KB page, 2MB page, as well as cache-line granularity (64 bytes). Redis-Rand suffers from the highest dirty data amplification with page granularity tracking, as high as 31X for 4KB pages and 5500X for 2MB pages, respectively (Table 2). In contrast, Redis-Seq has one of the lowest dirty data amplification, due to its sequential access pattern. All applications exhibit amplification (> 2) for page granularity tracking. In contrast, cache-line tracking results in a very small amplification (close to 1), suggesting that cache-line granularity would result in significant improvement.

Table 2: Dirty data amplification for different tracking granularities. The amplification is measured against the number of dirty *bytes*.

	Memory	Dirty data amplification		
Application	**(GB)**	**4KB page**	**2MB page**	**64B cache-line**
Redis-Rand	4	31.36	5516.37	1.48
Redis-Seq	0.13	2.76	54.76	1.08
Linear Regression	40	2.31	244.14	1.22
Histogram	40	3.61	1050.73	1.84
Page Rank	4.2	4.38	80.71	1.47
Graph Coloring	8.2	5.57	90.37	1.57
Connected Components	5.2	5.67	82.35	1.62
Label Propagation	5.6	8.14	95.00	1.85
VoltDB	11.5	3.74	79.55	1.17

2.2 Memory Access Patterns in Redis

The average dirty data amplification shown in Table 2 indicates that every application encounters pages that are not fully written. Next, we break down this amplification further by looking at access patterns of cache-lines within pages and their contiguity. We focus on two workloads at opposite extremes, with high and low amplification, Redis-Rand and Redis-Seq.

Spatial locality. Fig. 2 shows the cache-lines accessed within each page, as a CDF of pages with N accessed cache-lines, for N from 1 to 64 (full 4KB page). Pages have either a small number of cache-lines accessed (1-8 cache-lines), or all 64 cache-lines are accessed. Both workloads experience both types of pages, but Redis-Rand is skewed towards accessing a small number of cache-lines per page, while Redis-Seq is skewed towards accessing all cache-lines within each page. This result indicates that page granularity tracking and transfer is useful for some pages, but many pages can benefit from cache-line granularity.

Contiguous cache-lines. We define a segment as a group of contiguous cache-lines within a 4KB page that were accessed (read or written) in the same window. The length of a segment is given by the number of cache-lines in that segment. We count the segments

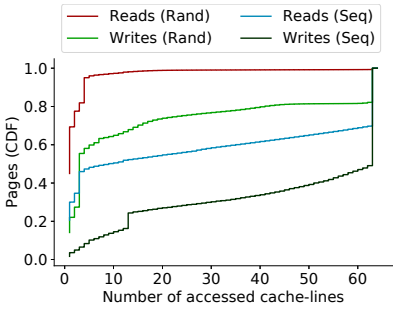


Figure 2: Accessed cache-lines in a page (Redis).

of length N, with N from 1 to 64 (a full page) and show the results as a CDF in Fig. 3 for Redis-Rand and for Redis-Seq. Most segments are of length 1 to 4 contiguous cache-lines for both workloads. For Redis-Seq, a large fraction of the segments are page-length, while for Redis-Rand, contiguous segments are short. Dirty cache-line contiguity is paramount for optimizing network transfer (§6.4).

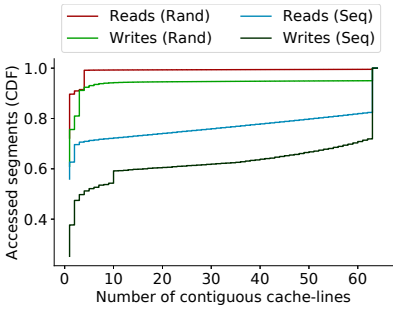


Figure 3: Contiguous cache-lines in a page (Redis).

2.3 Local Memory Coherence

Cache coherence protocols (e.g., VI, MSI, MESI, MOESI [60, 64]) ensure consistency between multiple cached copies of a memory location. When the CPU reads or writes a cache-line, it first requests the cache-line from a memory controller, which maintains access permissions for all cache-lines belonging to its physical memory. A cache also has to send back the data to the memory controller when it evicts a modified cache-line. Thus, for a given cache-line, the memory controller has excellent visibility to when the cache-line is being read or written.

Cache-coherent FPGAs (ccFPGAs) are connected to the CPU(s) using a point-to-point link that ensures memory coherence between a CPU-attached memory and an FPGA-attached memory (Fig. 5). The interconnect maintains coherence using a cache coherence protocol. Multiple ccFPGAs are expected to become available commercially or for research in the near future [3, 28, 41, 51], enabled by new interconnect standards, such as CXL [74] or CCIX [2]. These FPGAs can observe the CPU's local memory coherence events and use this information to enable new remote memory systems that do not suffer from the shortcomings of the current systems (§2.1).

3 OVERVIEW AND DESIGN PRINCIPLES

We propose a new class of software remote memory runtimes that rely on hardware cache coherence to speed up critical operations previously realized using virtual memory. Our main observation is that we can use the *unmodified* local hardware cache coherence protocol within a server to track applications' reads and writes. We designed and implemented Kona, a representative remote memory runtime based on this idea. Kona assumes the existence of new hardware primitives, which we describe in more detail in §4.

Below, we outline the key principles we use in our design and we discuss several benefits that Kona provides over a virtual-memory remote memory system.

Leverage hardware to track memory accesses. There is a semantic gap between applications and a remote memory runtime, which includes applications' memory accesses. This is generally resolved using expensive operations: for example, the remote memory implementation uses page faults to find out which pages the application has written (dirty data tracking). Our main observation is that the hardware *already* tracks memory accesses, through memory coherence. If the hardware exposed primitives that cached remote data and informed the software runtime of local modifications, the remote memory runtime could stop using inefficient virtual memory operations for these operations. Kona departs from state-of-the-art systems by relying on cache coherence to avoid page faults, write page faults and TLB shootdowns. Eliminating page faults from the critical path has the additional benefit that hardware prefetchers can prefetch more data, even from remote memory[4] and enables the CPU to avoid flushing the instruction pipeline, which happens in current systems due to the page fault.

Decouple data movement size from the virtual memory page size. As both application data and memory sizes are increasing, so are translation overheads. Therefore, it is natural for applications to improve performance by using large pages, but for applications that need to move data over the network, the drawbacks of dirty data amplification when using large pages outweigh the positives [77]. Kona tracks accessed and modified data at cache-line granularity, irrespective of the virtual memory page size. By decoupling the size of the tracked data from the page size, Kona enables applications to benefit from huge pages without suffering from data movement amplification (§6). Kona still relies on virtual memory for translation and protection, but can choose the data movement size between page and cache-line granularity.

Separate data and control path. Remote data access is on the applications' critical path, thus low-latency execution is paramount. Nevertheless, remote memory systems incur page faults on this critical path, significantly increasing the latency of a memory access [11]. For an efficient transparent remote memory runtime to be feasible, the low-latency data path operations need to be executed by the hardware. Kona expects such a hardware primitive to be available, replacing the need to rely on page faults for caching remote data. In contrast, control path operations are complex and require more flexibility, thus Kona implements them in software. Control path operations include setting up translation information for remote memory, enabling/disabling tracking, choosing policies to be executed by the hardware, resource management and error handling.

[4] A prefetch operation does not happen across a page fault, so current remote memory systems cannot benefit from the existing hardware prefetchers [43].

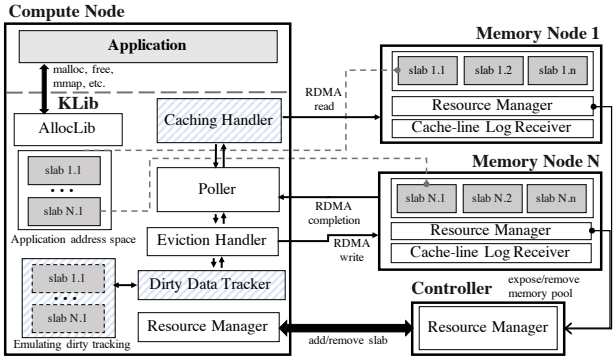


Figure 4: The Kona remote memory. Stripes indicate emulated components. Thick black rectangles represent different nodes in a rack. An application runs on a single compute node and accesses disaggregated memory on the memory nodes. Access to disaggregated memory is transparently realized by the KLib library. A rack controller allocates disaggregated memory at coarse granularity (large slabs).

4 KONA: COHERENCE-BASED REMOTE MEMORY

In this section, we describe Kona's design and implementation. Kona offers remote memory to applications transparently, without requiring source code changes to applications. We first describe the software runtime's high level architecture and components (§4.1). Kona assumes new hardware primitives, so we describe them next (§4.2) and we outline a reference hardware architecture that enables these primitives (§4.3). Finally, we discuss the runtime in more detail (§4.4) and provide a brief overview of Kona's failure mitigation options (§4.5).

4.1 Overview and Components

We show Kona's high-level software architecture in Figure 4. An application runs on a single compute node and can access disaggregated memory offered by one or more memory nodes. Disaggregated memory allocation is handled by a rack controller, which allocates memory at a coarse granularity, using large slabs. It does so off the critical path of the application. Each memory node has to register with the controller the amount of memory offered to applications. In our design, we assume the controller is a centralized entity managing the allocations [10], but a distributed approach is also feasible [36]. Similar to prior work, we assume each compute node has some amount of DRAM, which is used as a software cache for disaggregated memory [71].

The main part of the Kona runtime is an application library, KLib that hides all interactions with the controller, with the memory servers and with the new hardware primitives. KLib uses a Resource Manager to interact with the controller and pre-allocate disaggregated memory in large batches (slabs), which it maps in the

application's address space. In addition, KLib uses AllocLib, an allocation interposition library that handles fine-grained local memory allocations on the compute node. AllocLib interposes on applications' *malloc* and *mmap* calls and ensures that there is sufficient disaggregated memory available for the allocation.

KLib consists of three components that implement the three main remote memory operations: fetch, track, evict (§2). The Caching Handler fetches remote data that is not in the local DRAM cache when the application accesses it; the Dirty Data Tracker monitors data modified in the local DRAM cache; the Eviction Handler monitors the cache utilization and evicts pages to make room for new remote pages. These components rely on new hardware primitives. We discuss these primitives next. An additional component, the Poller, optimizes the RDMA communication with the controller and with the memory nodes, by polling for RDMA completions.

4.2 New Hardware Primitives

Current remote memory systems use virtual memory for the Caching Handler and for the Dirty Data Tracker. In essence, they use page faults to detect applications' reads and writes. This approach is often used in practice, but it incurs large overheads. For efficient remote memory, we need two new hardware primitives that provide the same two functions: (1) *cache-remote-data:* identify what data to fetch from remote memory and cache it in local memory; (2) *track-local-data:* identify what data has been modified locally and needs to be written back to remote memory, at fine granularity (i.e., cache-line).

The Eviction Handler copies dirty cache lines or pages to the remote host. While this operation can be realized on current hardware, it could also benefit from hardware acceleration. We propose a third, optional, hardware primitive: (3) *copy-dirty-data.*

With Kona, applications still use virtual memory for translation and protection, but Kona does not use virtual memory to provide remote memory. Next, we discuss a reference architecture for how such hardware support can be implemented.

4.3 A Reference Architecture Using FPGAs

Supporting remote memory caching and fine granularity dirty tracking in hardware using information from the cache coherence protocol requires changes to the CPU cache or memory controllers. However, such hardware changes are complex and intrusive, and have a long production cycle (if they ever become available). In this section, we propose an alternate mechanism [25], which we believe could be implemented sooner, when cache-coherent FPGAs become commercially available (§2.3).

The hardware architecture consists of an FPGA attached to the CPU using a coherent interconnect. Both the CPU and the FPGA have their own attached memories (CMem and FMem, respectively). The FPGA also exports a fake large physical address space, larger than FMem (called virtual FMem, or VFMem), which is not backed by local DRAM. Instead, the FPGA uses remote memory to back VFMem (Figure 5). The FPGA implements a memory agent that maintains a directory for VFMem, similar to current directories in the CPU. An application that accesses VFMem generates requests to the VFMem directory maintained by the FPGA. Thus, the FPGA can observe all the cache lines requested by the CPU from VFMem, and

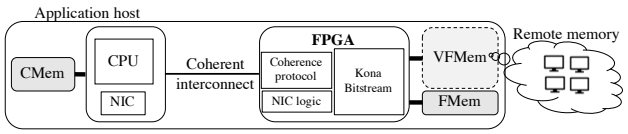


Figure 5: The Kona Architecture: An FPGA connected to a CPU through a coherent interconnect. Both the CPU and the FPGA have DRAM attached (CMem and FMem, respectively). The FPGA exposes fake physical memory to the CPU (VFMem), backed by remote memory.

fetch them from the disaggregated memory (the *cache-remote-data* primitive necessary for the Caching Handler). In addition, the FPGA can observe the cache-line writebacks, and track them in a bitmap for cache-line granularity dirty data tracking (the *track-local-data* primitive necessary for the Dirty Data Tracker).

This approach has the limitation that the FPGA cannot track CMem. To leverage this approach, we have to map all remote data in VFMem, to enable the FPGA to track accesses. All other memory for a process, such as thread stacks, global variables, executable pages, etc., are allocated from CMem.

The FPGA uses FMem as a cache for VFMem. The CPU never accesses FMem directly, but always accesses addresses in VFMem. Using VFMem for remote data results in two overheads: (1) accesses to the FPGA memory (FMem and VFMem) are slower than accesses to CMem, and (2) there is an additional translation step that the FPGA needs to perform from VFMem to FMem, even when the data is cached. FMem and VFMem are slower than CMem because of the limited interconnect bandwidth and because the directory logic is implemented in the FPGA. Eventually, this logic can be hardened, making its performance more competitive to a server NUMA system, where accessing a non-local socket is 1.5X slower than accessing the local socket [26]. Nevertheless, these overheads are much lower than current virtual memory and network overheads present in remote memory systems.

4.4 Remote Memory Operations in Kona

We describe how Kona works when our proposed hardware primitives are available. In §5, we describe how we simulate the FPGA hardware that is not yet available.

Allocating remote memory. The KLib Resource Manager requests remote memory from the rack controller (§4.1) and maps it in VFMem, logically pre-populating the memory. Since VFMem is a fake physical memory exposed by the FPGA, no physical memory is actually allocated at this time, only the page tables are set up and the pages are marked present.

Fetching remote data. In a state-of-the-art remote memory system, identifying data to fetch from remote memory is achieved using page faults (§2). Kona essentially replaces page faults with cache misses by mapping the remote data in VFMem and marking all pages as present. Thus, *when an application accesses data in VFMem, it will not incur any page faults* because the pages are already marked as present, but it will incur cache misses from all the CPU cache hierarchies. The CPU will thus send a cache-line request to the VFMem directory on the FPGA, which can fetch

the cache-line from the remote host on demand. Since pages remain mapped at the same location and with the same permissions, *this approach also avoids TLB invalidations and shootdowns*, which are otherwise incurred during eviction in a virtual memory based remote memory system.

Kona does not expose FMem directly to applications or to the OS, but uses it as a cache for VFMem. When the CPU accesses data from VFMem, the FPGA first checks if the data is cached in FMem, and if so, responds with the data. If the data is not cached, the FPGA fetches the data from the remote host, and decides whether to cache the data in FMem or not (based on how likely it is the data will be accessed again in the near future, or that nearby data – in the same page – will be accessed soon). FMem always caches entire pages. Moreover, the hardware prefetcher can request other cache lines likely to be accessed soon, which can cause the FPGA to prefetch the pages from remote memory. This is not possible in a virtual memory based remote memory, because page faults are serializing and the hardware prefetcher does not cross a page boundary [43].

Tracking dirty data. State-of-the-art remote memory uses write page faults for identifying what data has been modified locally (§2). Instead, with Kona, *we can avoid the write page faults* by tracking all cache-line write-backs that go to VFMem. The FPGA can identify which data has been modified without the page faults, and can do so at cache-line granularity. When the FPGA decides to write out dirty cache lines, it has to snoop them from CPU caches, in case the CPU has a newer copy of the data. Snooping is necessary because the FPGA only finds out about dirty data when the data is evicted from CPU caches and reaches memory.

Evicting dirty data. Kona uses a software log based on a ring buffer design similar to FaRM [29] to transfer dirty cache lines. We copy and aggregate the dirty cache-lines into the log, and use RDMA writes to transfer the log to the remote host. The Cache-line Log Receiver running on a thread on the remote host distributes the cache-lines from the received log into their locations and sends an acknowledgment to the application host. The process is asynchronous: the acknowledgment latency can be hidden by continuing to process more dirty cache-lines during the waiting time.

Address translation. The local host's page tables contain translations between the virtual addresses of a process to fake physical addresses in VFMem, as pages always remain mapped as present in VFMem. To discover the remote addresses of the missing pages, the FPGA uses a hashmap (*Remote translation*). The FPGA needs to implement additional metadata to keep track of which pages mapped in VFMem are present in the FMem cache (*Local translation*).

1) Remote translation. Upon a memory allocation, Kona stores metadata in a hashmap recording the remote memory addresses corresponding to each allocated slab in local memory. Kona allocates remote memory proactively in batches, so the allocation is not on the critical path. The remote allocation uses large sizes of one or multiple slabs. Kona uses a local memory allocator to split a large slab for smaller allocations on the client side. Kona stores the information in shared memory, with the FPGA being able to access it. The FPGA never updates the map, but it consults it when it fetches data from a remote host or when it writes dirty data back to a remote host.

2) Local translation. We design FMem as a 4-way set associative cache, with its block size equal to the page size. This approach is a good tradeoff that reduces the size of the metadata required to translate VFMem to FMem, while also ensuring that we keep the latency of a CPU memory access to VFMem low and enable a low eviction rate from the cache. Moreover, FMem always caches at page granularity instead of cache-line granularity, because the CPU hardware caches are sufficient to ensure that an application can benefit from temporal locality. The purpose for the FMem cache is to ensure that applications can also benefit from spatial locality.

4.5 Failures

Kona applications run on a single compute host, but access disaggregated memory located elsewhere in the rack. This is similar to prior work [36, 71]. Therefore, we consider three classes of possible failures: 1) the application or the compute host crashes; 2) the network is slow or unresponsive; 3) the disaggregated memory containing the application data fails. We discuss each failure mode in more detail.

1) Application or compute host failures. If the application or the application host fails, the application needs to be restarted, potentially on a different host. This is similar to today's monolithic server model. In this case, Kona does not add additional fault tolerance to applications that do not already provide it.

2) Network failures. A network failure or delay [59] is problematic because it can introduce timeouts in the cache coherence protocol, which has not been designed to handle large delays. The cache coherence protocol can result in a timeout due to slow or failed network operations, which triggers a machine check exception (MCE). There are two ways to address this problem: i) handle the MCE, which is possible using Intel's machine check architecture present on high-end servers [43]; or ii) move the page tables to FMem, which allows the FPGA to track the page table accesses and prefetch the remote data [25]. If the prefetch fails, the FPGA will mark the pages as not present, triggering a page fault, which enables the software to handle the failure, report it back to the application, wait until the network delay or outage is resolved, and/or notify an operator. This approach is inspired by translation-triggered prefetching [20].

3) Disaggregated memory failures. In case of memory failures, replication can ensure that the application data is still available. Kona can replicate the data during eviction, by sending the data to multiple replicas at the same time and waiting for all the acknowledgments. Kona reduces write amplification for each replica, so the network bandwidth improvement increases with the number of replicas. Adding more replicas can slow down eviction, but it rarely impacts application performance because eviction is not on the application critical path.

5 SIMULATING HARDWARE SUPPORT FOR KONA

We propose new primitives that can be implemented in a cache-coherent FPGA with attached memory. This hardware is not yet available, but it will be in the future (e.g., CXL-connected FPGAs), making our primitives feasible. In this section, we describe how the Kona components emulate the hardware primitives. The Caching Handler emulates *cache-remote-data* by instrumenting application reads and writes to remote memory. The Dirty Data Tracker emulates *track-local-data* by creating snapshots of the application's

pages cached in FMem. During eviction, the Dirty Data Tracker compares the application data with the snapshot to determine which cache-lines have been modified within each page.

Our evaluation focuses on applications, their access patterns and their locality characteristics. Our goal is to determine if our new hardware primitives can benefit real workloads and to compare with existing software solutions for disaggregated memory. To this end, we study the end-to-end benefit using an emulated implementation that relies on instrumenting application reads and writes. If the hardware were available, we would not need to instrument the application. In addition, we study each of the three remote memory operations in isolation: fetching remote data, tracking dirty data and evicting cached data. To do so, we had to develop our own tools. We are conservative in our assumptions, choosing the best case for the baselines. Next, we describe the two simulators we built, which allow us to measure each of the operations independently.

(1) Fetching remote data. We developed KCacheSim to simulate the *fetch from remote memory* operation without page faults. KCacheSim measures the average memory access time (AMAT) for applications running with Kona, Infiniswap and LegoOS. KCacheSim uses an existing cache simulator, Cachegrind [39], to determine the cache miss rates for each application from each level of the cache. Based on the cache miss rates, KCacheSim computes the AMAT. For Kona, we model the DRAM cache (FMem) as another level in the cache hierarchy, with a 4KB block size. For the baselines, we use main memory (CMem) instead of FMem. Using known average access latencies to remote memory (§2.1), we estimate the average access latency for all memory accesses in an application (local and remote). Our model includes the cost of the software stack in the remote memory access latency. Thus, we model a page fault as an increase in the transfer latency from remote memory. This is a conservative approach that favors the page fault based approach because it does not consider the impact of additional overheads that page faults cause: flushing the processor pipeline and hardware cache invalidations caused by the kernel mode execution. To determine the latency of a remote access for the baselines, LegoOS and Infiniswap, we run these systems to observe their latencies, including the overhead of the page faults. While this metric does not directly indicate application performance, it is a useful metric to understand how remote accesses with different latencies impact the application.

(2) Tracking dirty data. We developed KTracker to emulate Kona dirty data tracking at cache-line granularity by comparing snapshots of the application's memory in software (Fig. 6). KTracker uses *ptrace* to attach to a running process and create snapshots of its memory. Later, it diffs the application's memory with the copy to find out dirty cache lines. KTracker runs the application for a fixed amount of time, which gives us an indication of the application performance in real time, not simulated. KTracker updates its memory snapshot every second (a configurable parameter) and includes all accessed pages. While *fork* and *copy-on-write* can also be used to snapshot application's memory, we did not use this approach because we want to avoid causing additional page faults in the application. While the application runs at full speed during the execution, KTracker suffers from overheads in doing the diffs, which slows down the emulation (§6.3). KTracker can also run in write-protection mode, where it write-protects pages to track what pages have changed. This emulates a current remote memory system based on virtual memory, allowing us to compare the cache-line tracking in the same environment, with similar overheads, for a real apples-to-apples comparison.

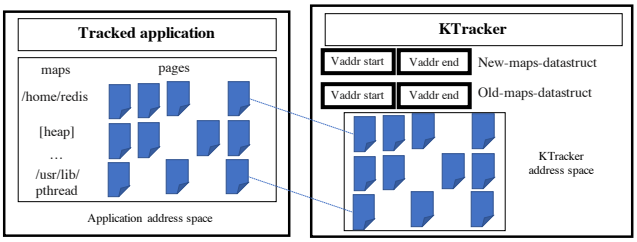


Figure 6: The KTracker simulator and its data structures.

5.1 Implementation

We implemented Kona as a C library that interposes on an application's memory allocation and uses a cooperative user thread for handling page faults [80]. The library has a total of 3.6k lines of code (LoC). The Kona server and controller run as separate daemons, and were implemented in 542 and 575 lines of C code, respectively.

Emulating hardware support for tracking dirty data. We implemented KTracker in C in 2.4k LoC. Kona uses a simplified version of KTracker to emulate cache-line dirty data tracking (200 LoC): for each page that is fetched from remote memory, we create a copy of the page that is used by the eviction thread to determine which cache-lines have changed when the page is evicted (Fig. 4).

RDMA eviction. To evaluate cache-line RDMA eviction, we implemented a microbenchmark that continuously writes dirty cache-lines or pages to a remote host using RDMA. We considered a few different optimizations for Kona cache-line RDMA eviction, as well as for the 4KB eviction baselines, and we kept those that were beneficial for each: (1) batching and linking multiple RDMA read or RDMA write operations together significantly improves the performance of the transfer; (2) unsignaled completions—we batch the completions as well as the requests, and only the last operation in a batch gets a completion from the NIC; (3) inline data turns out to not be beneficial for the size of transfers we considered (cache-line or 4KB) (4) using AVX instructions—copying data within the same host takes a lot of time but needs to be done because all RDMA reads and writes use buffers registered with the NIC; AVX instructions significantly reduce the overhead of the local copy.

6 EVALUATION

In this section, we evaluate Kona's end-to-end performance using emulation of the unavailable hardware primitives through application instrumentation. We compare Kona with a virtual memory remote memory system using a microbenchmark (§6.1). Next, we use our software tools and benchmarks to evaluate each of the three types of remote memory operations: fetching remote data (§6.2), tracking dirty data (§6.3), and evicting local data back to remote memory (§6.4).

Test-bed. We perform the RDMA experiments on a cluster of dual processor Skylake servers running at 2.2GHz with Mellanox Connect X5 cards connected through a 100Gbps RoCE switch. We run the simulations on CloudLab [31].

6.1 Microbenchmark Performance

In this section, we compare Kona with a virtual memory-based implementation (Kona-VM). First, we want to confirm that Kona-VM is on par with state-of-the-art virtual memory based systems. We compared Kona-VM with Infiniswap running Redis without any emulation or simulation on CloudLab instances c6220, where we could run Infiniswap successfully. Kona-VM is similar to or faster than Infiniswap by up to 60%.

Infiniswap has very high remote access latency (40μs). Most of the overhead comes from implementing Infiniswap as a block device and from using the bio layer (§2), with the rest distributed among small operations, such as finding and allocating VMAs, managing the page cache and the LRU list. In contrast, Kona-VM relies on handling page faults in user-space [80] and achieves lower latency than Infiniswap.

Next, we use emulation to compare Kona with Kona-VM, by instrumenting every remote memory access in the benchmark to provide the address, size and type of the memory access. Kona-VM is a good baseline to indicate the performance gains from our techniques, because Kona and Kona-VM use the same algorithms for data caching and eviction. Kona-VM uses virtual memory, while Kona emulates the proposed hardware primitives through benchmark instrumentation.

The benchmark allocates 4GB of remote memory *per thread*, and uses 1, 2, or 4 threads to read and write 1 cache-line in every page; each thread accesses distinct pages. As we increase the number of threads, the total amount of work increases. The benchmark reports the total execution time. Kona is faster than Kona-VM by 6.6X at 1 thread and by 4-5X for 2 and 4 threads when the benchmark runs with 50% local cache and eviction happens concurrently with the application execution (Figure 7). Kona only writes the dirty cache-lines to the remote host, while Kona-VM has to write entire pages. Both Kona and Kona-VM use the same algorithm and make the same decisions about which pages to evict. This ensures that the results reflect the difference between page and cache-line granularities and not a difference in eviction algorithms. Kona can copy the data to remote memory directly from FMem; it does not need an additional copy to CMem first. Thus, no NUMA penalties are incurred.

Next, we evaluated the benchmark with all the initial data in remote memory, but without eviction from the DRAM cache. Here, Kona-NoEvict is faster than Kona-VM-NoEvict by 3-5X. Kona-VM incurs two page faults for caching a remote page. The first is to fetch the page from remote memory, and the second, minor page fault removes the write-protection on the page, marks the page dirty and enables the write. Kona avoids both page faults. We also ran a version of Kona-VM that avoids write-protection (NoWP), so it only incurs one page fault. This version cannot track dirty pages so it is incomplete, yet it is still slower than Kona-NoEvict by 1.2-2.9X.

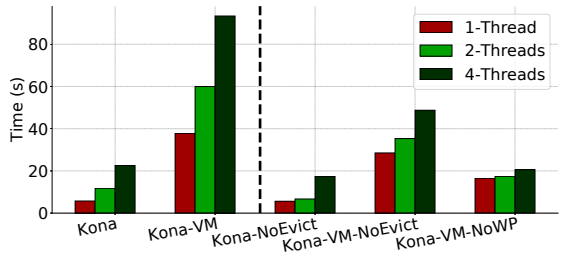


Figure 7: Kona and Kona-VM.

6.2 Fetching Remote Data

In this section, we evaluate the remote data fetch operation. We use KCacheSim to study the average memory access time (AMAT) for applications accessing remote memory as a function of the available cache size, cache associativity and cache block size. Finally, we measure the overhead of the simulation.

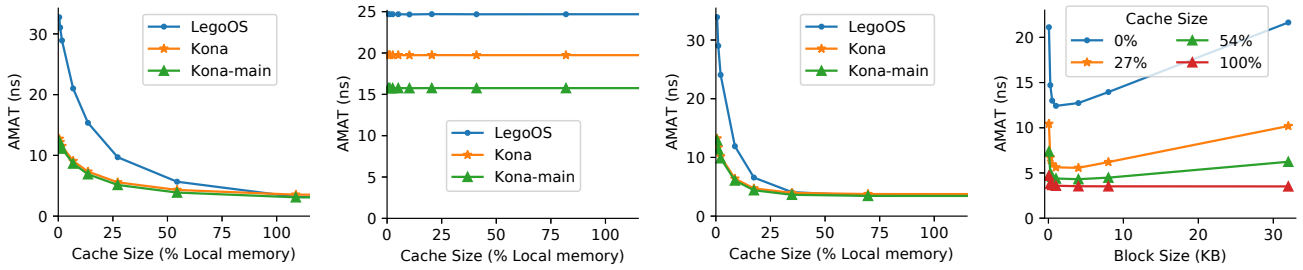
KCacheSim simulates Kona's Caching Handler component and compares it to remote access based on virtual memory. KCacheSim models remote access latencies based on our measurements using real RDMA hardware and different memory hierarchies for each system. For Kona, the memory hierarchy includes hardware caches, FMem (NUMA memory with higher latency) and remote memory. We also evaluate Kona-main, a version of Kona where the data is cached in CMem, thus avoiding the NUMA overheads present in Kona. This shows the best performance that Kona can achieve if it could track CMem, not only FMem (likely via processor architecture modifications).

For Infiniswap and LegoOS, the memory hierarchy includes hardware caches, CMem (locally attached DRAM) and remote memory. We measured remote access latencies in these systems running on real hardware (*not* in simulation), including page fault overheads and we use these measurements for the simulation (10 μs for LegoOS and 40 μs for Infiniswap). Infiniswap is consistently worse than LegoOS by 2.3-3.7X, so we do not show it on the graphs for better visibility. We also omit Kona-VM, which achieves similar remote access latency with LegoOS, resulting in similar AMAT.

LegoOS is orthogonal to Kona and to the main ideas in this paper. In fact, cache-line granularity and avoiding page faults could also be applied to LegoOS to improve remote memory caching and eviction. Here, we use LegoOS as a baseline because it achieves lower remote access latency than Infiniswap.

CMem has lower latency than FMem, while Kona's remote accesses are faster than the baselines' remote accesses because Kona avoids page faults. Thus, there is a tradeoff between lower local latencies and lower remote latencies. We experimented with multiple classes of applications (map-reduce, graph analytics, key-value stores), to explore these tradeoffs. Our model does not consider network congestion, but we use the same model for Kona as well as for the baselines. Our simulations are with memory prefetching turned off, so our results are conservative for Kona, which can benefit from hardware prefetching even when the data is in remote memory.

(1) AMAT. For large cache sizes, close to 100% of application peak resident set sizes, all systems perform similarly because the number



(a) Redis Rand (b) Linear Regression (c) Graph Coloring (d) Redis Rand - Data fetch size

Figure 8: Simulating remote data fetch

of remote accesses is small. However, the AMAT increases quickly for smaller caches, as the applications incur more (expensive) remote memory accesses. As Fig. 8 shows, Kona makes disaggregated memory possible, as the AMAT increases much more slowly compared to the software systems. When only 25% of the data is cached, which is not unrealistic for disaggregated memory, Kona achieves 1.7X and 5X lower AMAT than LegoOS and Infiniswap, respectively. The one exception is Linear Regression (Fig. 8b), where the memory access latency is almost constant irrespective of local cache size. This behavior is due to the workload's streaming access pattern, where there is almost no data reuse and hence little use for a local cache. Kona incurs overhead from caching remote data in FMem, due to NUMA effects, as shown by the comparison with Kona-main. We measured the worst overhead for Linear Regression (25%), while Redis and Graph Coloring incur only 2-13% higher AMAT due to NUMA effects.

(2) Associativity and cache block size. We found that the associativity does not significantly impact overall latency. We considered cache block sizes ranging from cache-line (64B) up to 30KB for different percentages of the local cache size (Fig. 8d). Kona can fetch remote memory at any granularity larger than a cache-line. However, we found that small block sizes did not fully exploit the available spatial locality in the applications (Fig. 8d). In contrast, large block sizes increased conflict misses and latency. For almost all of our workloads, we found that 1KB block size achieved the lowest AMAT. 4KB size increased the AMAT by a small margin, thus we made the decision to use a block size of 4KB (page granularity) for remote data fetch, which simplifies metadata management. In contrast, we use cache-line granularity for dirty data tracking.

(3) Simulation overhead. We measured KCacheSim's simulation overhead by running Redis with the same workload both natively and in simulation. Redis has 43X lower throughput running in simulation.

6.3 Tracking Dirty Data

We evaluate coherence-based cache-line granularity dirty data tracking without page faults and compare it to the virtual memory based 4KB granularity write-protection approach. We developed KTracker (§5) to simulate cache-line tracking. KTracker allows us to understand (1) how much coherence-based remote memory reduces the dirty data amplification using cache-line instead of 4KB-page tracking; and (2) coherence-based remote memory's speedup due to avoiding write page faults. We also evaluate (3) the KTracker simulation overhead. KTracker simulates the Dirty Data Tracker component and compares it to dirty data tracking using virtual memory based write-protection with a real-time window of 1-second. KTracker tracks dirty data only locally, without using the network.

(1) Tracking granularity. First, we measured the dirty data amplification with cache-line granularity tracking. We show the 4KB-page amplification *relative to cache-line tracking* in Figure 9. We show Redis-Seq and Redis-Rand, which represent the more extreme workloads. The Redis-Seq workload finishes faster than Redis-Rand, so it requires fewer 1 second windows. The first 10 windows of the experiment are the server startup and initialization, so they look similar for both workloads. Cache-line granularity reduces the amplification for both Redis-Rand and Redis-Seq, by 2-10X and by 2X, respectively. As expected, the random workload experiences higher amplification and thus the benefit from cache-line granularity is higher.

Other workloads experience an amplification between Redis-Rand and Redis-Seq. The graph analytics and the map-reduce workloads perform random access and sequential access, respectively, with a cyclic amplification behavior.

We measure the amplification up until the process exit, when KTracker gets an exit notification. The last window contains the normal process tear-down, which includes a small number of writes with high amplification (e.g., main() return value). We do not consider this last window in the results reported because it skews the average amplification in the favor of cache-line granularity (e.g., we report a 2-10X amplification for Redis-Rand, without including the 30X amplification in the last window).

(2) Tracking speedup. We use KTracker to measure the coherence-based dirty data tracking speed-up compared to 4KB-page write-protection and report the results in Fig. 10. The speedup ranges from 1% (Redis-seq and Histogram) to 35% (Redis-rand).

(3) Simulation overhead. To measure the simulation overhead, we ran the Redis server with the same workload natively and with KTracker. In both cases, we use a memtier client [4] that runs natively. The simulated Redis server experiences 60% lower throughput, 95% of which is caused by copying and comparing the application's memory. 5% of the overhead is caused by using ptrace. Next, we compared the native and the simulated executions to determine if there are any unexpected side-effects of running Redis with

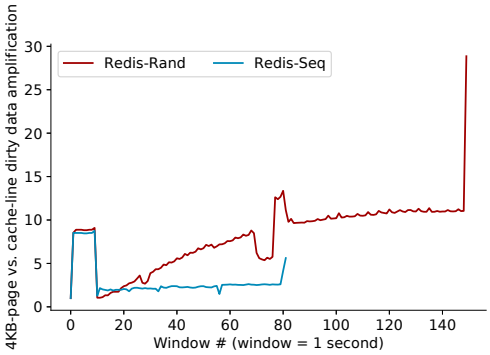


Figure 9: Dirty data amplification reduction.

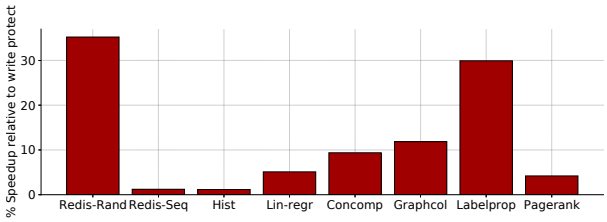


Figure 10: Speedup relative to write-protection.

KTracker. Unsurprisingly, the running time of the application is longer due to the pauses that KTracker introduces. In addition, the number of context switches increased by 62%, while CPU utilization for Redis decreased by 57% with KTracker. However, we did not notice any other significant differences. In particular, the numbers of page faults, L3 cache misses and TLB misses were similar between the two executions.

6.4 Evicting Local Data Back to Remote Memory

We evaluate cache-line granularity eviction and compare it to traditional virtual memory based eviction (4KB page granularity). Kona-VM evicts 4KB pages and writes all *dirty pages* to the remote hosts using RDMA one-sided verbs [49], after having copied them from the application's address space to RDMA-registered buffers. Kona also evicts 4KB pages, but writes only the *dirty cache-lines* to the remote hosts, an ability enabled by Kona's cache-line granularity dirty data tracking. Like Kona-VM, Kona also needs to copy the dirty data to RDMA-registered buffers before using RDMA verbs, so it uses the buffers to aggregate multiple dirty cache-lines, even from different pages, into bigger chunks that can be written together to the remote host. We call the buffer with the aggregated cache-lines the *CL log*.

For this experiment, we use a microbenchmark that continuously writes N cache-lines out of each 4KB page in a 1GB region, for values of N between 1 and 64 (full 4KB page). The benchmark then writes the dirty data to a remote host using RDMA.

We evaluate two scenarios: *contiguous* or *alternate* (representing random) dirty cache-lines in a page. Copying contiguous dirty cache-lines is more efficient due to higher cache locality enabled by automatic next cache-line prefetching. We measure the total time to complete the transfer, including any necessary acknowledgment from the remote host (only for the CL log), and compute the *goodput* for each situation based on the number of dirty cache-lines. We define goodput as the amount of *useful* data (i.e., the dirty cache-lines) transferred over the network during the measurement time.

Kona's cache-line log (CL log) achieves 4-5X higher goodput than Kona-VM's 4KB writes for 1-4 contiguous dirty cache-lines (Fig. 11a) and 2-3X higher goodput for 2-4 random cache-lines (Fig. 11b). Kona-VM has lower goodput than Kona because Kona-VM uses 4KB RDMA writes, which transfer more data than necessary over the network (e.g., for 1 dirty cache-line, 4032 bytes are actually clean, but 4096 bytes are transferred, not including packet headers).

If dirty cache-lines are contiguous, Kona is always better than Kona-VM, or on par when the whole page is dirty. Kona achieves lower goodput than Kona-VM only for more than 16 discontiguous dirty cache-lines, which is rare in real applications. Even pages that have many dirty cache-lines show some contiguity. As we have shown in §2.2, pages often have 1-8 dirty cache-lines.

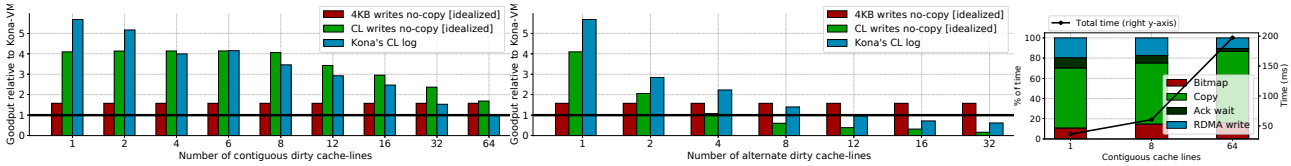
Unlike Kona-VM, Kona needs a thread running on the remote host to unpack the log of dirty cache-lines and write them at their proper destination in the remote memory. The remote thread reads sequential cache-lines from the log received from the application host and writes each cache-line at its proper address, then sends an acknowledgment. The overhead of the remote thread is small, consisting of a few memory reads and writes.

We show a breakdown of the Kona cache-line eviction performance in Figure 11c. Most of the time is spent copying the data to the RDMA buffer (*Copy*). 15-20% of the time is spent on the RDMA operations (*RDMA write*), with another 15-20% checking a bitmap to determine which cache-lines are dirty (*Bitmap*). Finally, there is a small amount of time spent waiting for an acknowledgment from the remote host, which has to unpack the aggregated dirty cache-lines (*Ack wait*).

Idealized baselines. In figures 11a and 11b we also show two idealized baselines that require no copy to an RDMA buffer: *4KB writes no-copy* and *CL writes no-copy*. The idealized baselines use RDMA writes to copy the data to the remote host, at 4KB page granularity and cache-line granularity, respectively. For these baselines we use local buffers that are already registered for RDMA, so no copy is necessary locally. Similarly, the remote addresses are registered, so there is no remote thread unpacking cache-lines. These baselines cannot be used directly in a remote memory system, because the application's address space is not registered for RDMA and a copy to a separate buffer is required, but we include them for comparison.

4KB writes no-copy always achieves 1.5X higher goodput than Kona-VM, which needs the additional local 4KB copy for each page. *CL writes no-copy* work well for a small number of contiguous cache-lines, but do not work well when dirty cache-lines are discontiguous or when a large part of the page is dirty, because many small RDMA operations need to be issued.

In contrast, Kona aggregates dirty cache-lines in the RDMA buffer, whether they are contiguous or not, and can issue fewer RDMA writes, of larger size, resulting in better network utilization. We use linking and batching to optimize both Kona and the baselines, but Kona is more efficient because it submits a single request to the NIC for the whole log. We also experimented with



(a) Goodput for contiguous dirty cache-lines **(b) Goodput for alternate dirty cache-lines** **(c) CL log time breakdown**

Figure 11: Eviction goodput using cache-line (CL) granularity

using the scatter-gather NIC functionality, but the performance was consistently worse than Kona (not shown), due to inefficiencies in gathering many different entries.

7 DISCUSSION

Disaggregated memory provides multiple benefits: it improves memory utilization, decreases memory over-provisioning and allows independent scaling of memory and compute hardware. For example, applications running in a datacenter might need TB of memory altogether, but not so much CPU. Independent scaling through disaggregated memory allows the datacenter operator to add more memory, without having to add more CPUs that will end up being underutilized.

These benefits cannot be achieved by increasing the amount of memory in a single host. Memory utilization in current datacenters is already low [63, 66, 78]. Increasing the amount of memory on individual hosts worsens datacenter memory utilization, bringing up both capital and operating expenditures. In addition, memory cannot be scaled up arbitrarily on a single host due to the limited number of DIMM slots, and CPU grade requirements (premium CPUs are needed for large memory machines, which significantly increases cost).

Our approach requires adding a cache coherent FPGA to each application host, which is an additional cost. However, we expect this cost to be small compared to the overall cost savings brought by enabling disaggregated memory – by improving memory utilization and enabling independent scaling of CPUs/memory.

Similar to prior work on systems for disaggregated memory (LegoOS, Infiniswap, etc.), Kona targets applications that run on a single node. Kona does not offer distributed shared memory. Applications running on the same host or on different hosts can share data through files, but they require additional coordination if they access the data concurrently.

Applications never allocate disaggregated memory directly. This is done transparently by Kona. Kona exposes disaggregated memory as if it is local memory to applications through a large fake physical address space, and transparently allocates and populates remote memory with the application data created locally.

8 RELATED WORK

To our knowledge, we are the first to propose coherence-based remote memory. We discuss other remote memory systems, techniques for tracking applications' access patterns, compiler support for sub-page memory access tracking and performing computation at memory nodes in a disaggregated memory environment.

Remote memory systems. Works on distributed shared memory [16, 19, 48, 69, 70] provide shared memory and cache coherence across hosts. In contrast, Kona leverages local cache coherence within a single host to expose remote memory to legacy applications transparently. The availability of low-latency networking makes remote memory practical. Recently, we are seeing a resurgence of research in this field [10, 15, 36, 57, 71]. However, these works, including disk swapping [1, 6, 44], rely on page faults and page-based tracking, which limits their performance. Some remote memory systems [29, 30, 61, 67] use an object-based interface that avoids the virtual memory subsystem's overhead, but these systems require modifications of the application code. Meanwhile, Kona avoids virtual memory overhead by using the local cache coherence traffic via the cache-coherent FPGA to access remote data while also remaining transparent to the applications. Hence, cache-coherent FPGAs provide an alternative solution to the remote memory problem. Kona's use of FPGAs is in line with the emerging trend of increasingly using FPGAs in the datacenter [27, 65], albeit for different purposes, such as accelerating applications [12–14, 22, 35, 38, 47, 55, 62, 73, 75, 76], smart NICs [34, 58], and allowing multi-tenancy [46, 54, 83].

Tracking application access patterns. Prior works have explored sub-page granularity memory access tracking by using (1) specific APIs [29, 30, 61, 67], (2) source code annotations requiring application modification [61], (3) run-time techniques to track reads and writes [5, 23, 53], (4) architectural simulations [21, 68], and (5) hardware support for sub-page protection [17, 40]. Intel introduced Page Modification Logging (PML), which logs modified pages in hardware and informs the hypervisor of dirty pages in batches of 512 pages [42]. PML reduces the overhead of dirty data tracking, but continues to rely on page granularity. These approaches trade-off generality, tracking granularity, and application performance based on the specific use-case. In this work, we use a suite of techniques to achieve cache-line granularity tracking of applications' access patterns without experiencing the slowdowns that other tools incur.

Compiler support. A software-only solution could use language or compiler support to track applications' accesses at finer granularity than a page. Unfortunately, there is no off-the-shelf transparent software-only solution that we can compare with Kona. If there existed such a solution, there would be tradeoffs. The software solution sacrifices generality and cannot support arbitrary code (e.g., the guest kernel in a VM). In contrast, our solution requires new hardware, but it is not limited to a specific language or compiler.

Computation at memory nodes. Kona uses a remote thread at the disaggregated memory to unpack aggregated dirty cache-lines and distribute them to their memory addresses. Other proposals

for disaggregated memory also make use of a remote thread for memory management, performing even more operations than Kona. For example, Semeru [81] uses remote threads for tracing pointers in garbage collection.

9 CONCLUSION

We introduced coherence-based remote memory, a new class of remote memory that uses the local host's cache coherence mechanisms to track applications' memory accesses. We identified two main hardware primitives needed to enable coherence-based remote memory. We designed and implemented Kona, a representative software runtime that uses these primitives to reduce dirty data amplification and to improve network utilization and application performance. In addition, we developed new software tools to evaluate coherence-based remote memory using emulation and simulation with microbenchmarks and real applications. Using these tools, we show that coherence-based remote memory improves the average memory access time by 1.7-5X and reduces dirty data amplification by 2-10X, compared to state-of-the-art systems. We conclude that coherence-based remote memory is a promising approach to building efficient disaggregated memory.

ACKNOWLEDGMENTS

We are grateful to our shepherd, Steve Blackburn, to the anonymous reviewers, and to Nadav Amit, Mihai Budiu, Jon Howell, Chris Rossbach, Lalith Suresh and Amy Tai for their thoughtful feedback.

REFERENCES

[1] Balance LRU lists based on relative thrashing. https://lwn.net/Articles/690069/.
[2] CCIX. https://www.ccixconsortium.com.
[3] Enzian, a research computer built by the Systems Group at ETH Zürich. http://www.enzian.systems/index.html.
[4] memtier benchmark: A high-throughput benchmarking tool for redis and memcached. https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/.
[5] Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[6] Reconsidering swapping. https://lwn.net/Articles/690079/.
[7] Redis: open-source, in-memory data structure store. https://redis.io.
[8] VOLTDB. https://www.voltdb.com/.
[9] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.
[10] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference (ATC)*, 2018.
[11] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.
[12] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, 35(21), 2019.
[13] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics*, 33(21), 2017.
[14] Mohammed Alser, Taha Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs. *Bioinformatics*, 2020.
[15] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *European Conference on Computer Systems (EuroSys)*, 2020.
[16] Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.
[17] Apple. How We Ported Linux to the M1. https://corellium.com/blog/linux-m1.
[18] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.
[19] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 1990.
[20] Abhishek Bhattacharjee. Translation-triggered prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
[21] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
[22] M. Blott and K. Vissers. Dataflow architectures for 10 Gbps line-rate key-value-stores. In *IEEE Hot Chips 25 Symposium (HCS)*, 2013.
[23] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *International Conference on Virtual Execution Environments (VEE)*, 2012.
[24] Irina Calciu, Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory, February 2021. https://github.com/project- kona/asplos21-ae.
[25] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project PBerry: FPGA Acceleration for Remote Memory. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.
[26] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
[27] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *International Symposium on Microarchitecture (MICRO)*, 2016.
[28] Convey Computer. The Convey HC-2 Computer. Architectural Overview. https://www.micron.com/~/media/documents/products/white-paper/wp_convey_hc2_architectual_overview.pdf, 2012.
[29] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation (NSDI)*, April 2014.
[30] Aleksandar Dragojević, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.
[31] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, 2019.
[32] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016.
[33] Gen-Z draft core specification—december 2016. http://genzconsortium.org/draft-core-specification-december-2016.
[34] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA: An open platform for teaching how to build Gigabit-rate network switches and routers. *IEEE Transactions on Education*, 2008.
[35] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. Accelerating Arithmetic Kernels with Coherent Attached FPGA Coprocessors. In *Design, Automation & Test in Europe (DATE)*, 2015.
[36] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with Infiniswap. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
[37] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, August 2016.
[38] Zhenhao He, David Sidler, Zsolt István, and Gustavo Alonso. A flexible k-means operator for hybrid databases. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.
[39] Intel. Cachegrind. https://valgrind.org/docs/manual/cg-manual.html.
[40] Intel. EPT-based Sub-Page Permissions. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.
[41] Intel. Intel Xeon+FPGA Platform for the Data Center. http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf.

[42] Intel. Page Modification Logging for Virtual Machine Monitor White Paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf.

[43] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. November 2020.

[44] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *International Symposium on Memory Management (ISMM)*, 2002.

[45] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2015.

[46] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, 2018.

[47] Maysam Lavasani, Hari Angepat, and Derek Chiou. An FPGA-based in-line accelerator for Memcached. *IEEE Computer Architecture Letters*, 2014.

[48] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, November 1989.

[49] libibverbs. http://www.rdmamojo.com/2012/05/18/libibverbs.

[50] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, February 2012.

[51] Liu Ling, Neal Oliver, Chitlur Bhushan, Wang Qigang, Alvin Chen, Shen Wenbo, Yu Zhihong, Arthur Sheiman, Ian McCallum, Joseph Grecco, Henry Mitchel, Liu Dong, and Prabhat Gupta. High-performance, Energy-efficient Platforms Using In-socket FPGA Accelerators. In *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2009.

[52] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[53] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[54] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A Hypervisor for Shared-Memory FPGA Platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[55] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[56] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, May 2010.

[57] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *USENIX Annual Technical Conference (ATC)*, 2020.

[58] Mellanox. Mellanox Innova™ IPsec 4 Lx Ethernet Adapter Card User Manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_Innova_IPsec_4_Lx_Ethernet_Adapter_Card_User_Manual_rev_1_3.pdf.

[59] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference (IMC)*, 2018.

[60] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence, second edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[61] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference (ATC)*, July 2015.

[62] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[63] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. Quantifying Memory Underutilization in HPC Systems and Using It to Improve Performance via Architecture Support. In *International Symposium on Microarchitecture (MICRO)*, 2019.

[64] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *International Symposium on Computer Architecture (ISCA)*, 1984.

[65] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[66] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, 2012.

[67] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[68] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[69] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[70] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1994.

[71] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, 2018.

[72] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[73] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[74] Navin Shenoy. A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data.

[75] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppioDB: A hardware accelerated database. In *International Conference on Management of Data (SIGMOD)*, 2017.

[76] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NERO: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2020.

[77] Mario Smarduch. Enhanced Live Migration For Intensive Memory Loads. https://events.static.linuxfound.org/sites/events/files/slides/CloudOpen-Japan-2015.pdf.

[78] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *European Conference on Computer Systems (EuroSys)*, 2020.

[79] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[80] Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt.

[81] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–280, November 2020.

[82] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *International Conference on Very Large Data Bases (VLDB)*, 10(6), February 2017.

[83] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.