Using Local Cache Coherence for Disaggregated Memory Systems

Irina Calciu
Graft

M. Talha Imran
Google

Ivan Puddu
ETH Zürich

Sanidhya Kashyap
EPFL

Hasan Al Maruf
University of Michigan

Onur Mutlu
ETH Zürich

Aasheesh Kolli
Google

ABSTRACT

Disaggregated memory provides many cost savings and resource provisioning benefits for current datacenters, but software systems enabling disaggregated memory access result in high performance penalties. These systems require intrusive code changes to port applications for disaggregated memory or employ slow virtual memory mechanisms to avoid code changes. Such mechanisms result in high overhead page faults to access remote data and high dirty data amplification when tracking changes to cached data at page-granularity. In this paper, we propose a fundamentally new approach for disaggregated memory systems, based on the observation that *we can use local cache coherence to track applications' memory accesses transparently, without code changes, 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 and reduces dirty data amplification[1].

1 INTRODUCTION

Modern data centers suffer from memory over-provisioning and under-utilization, with reports of memory utilization stagnating around 65% [78]. Disaggregated memory [11, 32, 37, 50, 79, 82] can address these issues, but systems that enable adoption of disaggregated memory either cause intrusive code changes to existing applications, require a change in the programming model for new applications or come with high performance penalties to achieve these changes transparently, without application changes.

Transparent systems use various kernel subsystems [10, 15, 36, 57, 72] or redesign the kernel altogether [71] to avoid application code changes. 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. 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, 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, 67] 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 disaggregated memory is error-prone and requires expensive engineering resources and expertise.

Our key insight is that the local cache coherence protocol can provide better support for disaggregated memory, by transparently tracking applications' memory accesses at cache-line granularity, without page faults. We describe a reference architecture that provides the necessary hardware primitives using cache-coherent field programmable gate arrays (§4.2), which we expect to become available with the adoption of CXL-based platforms [74]. We designed and implemented Kona, a software disaggregated memory system that rethinks the design of each of the three disaggregated memory functions performed by virtual memory in current systems (fetching remote data, 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.

To evaluate Kona's benefits and overheads without the hardware support, we developed several tools (§4.4) that allow us to simulate or emulate the necessary hardware primitives. Thus, we show that Kona improves average memory access time when accessing disaggregated memory by 1.7X compared to LegoOS [71], and reduces write amplification compared to page-based systems by 2-10X, by using cache-line granularity.

In summary, we make the following contributions:

- We show that current systems for disaggregated memory result in large overhead and dirty data amplification (§2).
- We propose new hardware primitives for disaggregated memory based on the local cache coherence protocol (§4.2) and

[1]This paper is based on an earlier work: Rethinking Software Runtimes for Disaggregated Memory, in Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, April 19-23, ACM 2021. https://doi.org/10.1145/3445814.3446713

we design and implement Kona, a software system that uses the new hardware primitives for efficient execution (§4).
- We design and implement multiple emulation and simulation tools (§4.4) and we use them to evaluate Kona (§5).

2 DISAGGREGATED MEMORY SYSTEMS TAXONOMY AND TRADEOFFS

We provide a taxonomy of disaggregated memory systems, and discuss their tradeoffs. We classify the various systems based on the access and tracking granularity, programability and the mechanism employed to offer access to disaggregated memory (Table 1).

Table 1: Taxonomy of disagg. memory systems.

Disagg. 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 disaggregated memory systems offer access to disaggregated memory transparently by using virtual memory [10, 36, 45, 72] to cache remote pages into a local software-managed DRAM cache, trading off the granularity of access and application performance. These systems use page faults to **fetch data from a remote host** using a custom page fault handler. They use write page faults to **track dirty data**: a page is initially marked as read-only when it is first fetched into the local DRAM cache and marked *dirty* if the application modifies it, by triggering a page fault to lift the write-protection on the page. 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, while dirty pages need a writeback to the remote host.

By relying on the virtual memory mechanisms, disaggregated memory operations in page-based systems suffer from high overhead. (1) Fetching remote data incurs large penalties due to page faults and TLB invalidations. In addition, 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. (2) Dirty data tracking and evictions suffer from large overheads associated with write page faults, which requires stopping the application to modify page tables and invalidate the TLBs. For example, we measured a 35% decrease in throughput for Redis [7] due to write page faults. (3) Tracking pages results in high dirty data amplification, which is the ratio of data marked as dirty using the tracking granularity to the actual number of bytes written by the application. 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. Table 2 shows the dirty data amplification in several applications measured using dynamic binary instrumentation with Intel Pin [5].

In contrast to page-based systems, *object-based* disaggregated memory systems provide fine-grain access to remote data using data structures or key-value abstractions [29, 30, 61, 67]. To achieve good performance, these systems trade off programability by relying on semantic information communicated by the programmer and require intrusive code changes to port legacy applications.

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
Page Rank	4.2	4.38	80.71	1.47
Label Propagation	5.6	8.14	95.00	1.85
VoltDB	11.5	3.74	79.55	1.17

We leverage the main benefits from both types of systems by proposing new hardware support based on the local host's cache coherence mechanism. Our system, Kona, maintains application transparency, while accessing memory at a finer-granularity (cache-line) and can support legacy applications without modifying them.

3 DESIGN PRINCIPLES

We propose a new class of software systems for disaggregated memory that uses the *unmodified* local host's hardware cache coherence protocol to transparently track reads and writes performed by an application and thus speed up critical operations previously realized using virtual memory. Below, we outline the key design principles we employed and we discuss the benefits that our approach provides over a page-based disaggregated memory system.

Leverage cache coherence to track memory accesses. Applications that access disaggregated memory suffer from a semantic gap that is either resolved explicitly by the application through code changes or through expensive virtual memory operations. 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 for these operations. Thus, we can avoid page faults, write page faults and TLB shootdowns.

Decouple data movement granularity from the 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]. Our approach uses cache-line granularity to track data accesses and movement, irrespective of the virtual memory page size. By decoupling the size of the tracked data from the page size, we enable applications to benefit from huge pages without suffering from data movement amplification (§5).

Separate data and control paths. Remote data access is on an application's critical path, thus low-latency execution is paramount. Nevertheless, systems for disaggregated memory 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. In contrast, control path operations are complex and require more flexibility, thus our approach is to implement them in software. Control path operations include setting up translation information for disaggregated memory, enabling/disabling

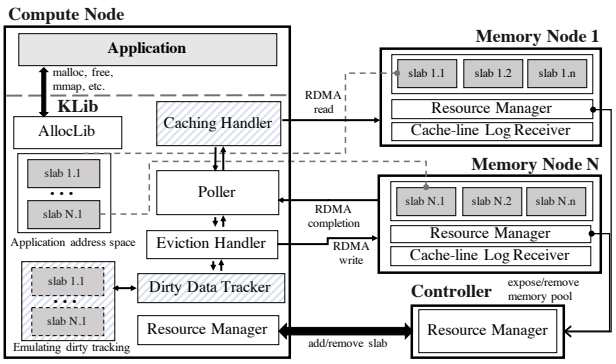


Figure 1: The Kona runtime system. Stripes indicate emulated components. Thick black rectangles represent different nodes in a rack. An application runs on a single compute node and accesses disaggregated memory on the memory nodes. Access to disaggregated memory is transparently realized by the KLib library. A rack controller allocates disaggregated memory at coarse granularity (large slabs).

tracking, choosing policies to be executed by the hardware, resource management and error handling.

4 COHERENCE-BASED DISAGGREGATED MEMORY

In this section, we describe the building blocks for coherence-based disaggregated memory and we present the design and implementation of Kona, a runtime system that showcases these ideas.

We show Kona's high-level software architecture in Figure 1. 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 disaggregated 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, which we describe next. An additional component, the Poller, optimizes the RDMA communication with the controller and with the memory nodes, by polling for RDMA completions.

4.1 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 disaggregated memory, we need two new hardware primitives that provide the equivalent functions: (1) *cache-remote-data:* identify what data to fetch from disaggregated 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 disaggregated 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 access to disaggregated memory. Next, we discuss a reference architecture for how such hardware support can be implemented.

4.2 A Reference Architecture Using FPGAs

We propose a hardware architecture that consists of an FPGA attached to the CPU using a coherent interconnect (Figure 2). Both the CPU and the FPGA have their own attached memories (CMem and FMem, respectively). In addition, the FPGA exports a large fake physical address space, larger than FMem (called virtual FMem, or VFMem), backed by the disaggregated memory instead of the local DRAM. 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, allowing the FPGA to observe all the cache lines requested by the CPU from VFMem, and to 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

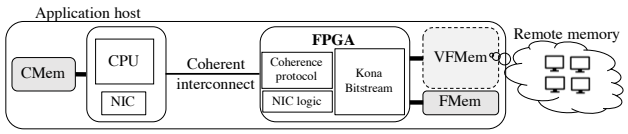


Figure 2: The Kona Architecture: An FPGA connected to a CPU through a coherent interconnect. Both the CPU and the FPGA have DRAM attached (CMem and FMem, respectively). The FPGA exposes fake physical memory to the CPU (VFMem), backed by remote memory.

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 disaggregated memory systems.

4.3 Disaggregated Memory Operations in Kona

We describe how Kona works when our proposed hardware primitives are available. In §4.4, 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) 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 disaggregated memory system, identifying remote data to fetch 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 sends 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 disaggregated memory system based on virtual 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.

4.4 Simulating Missing Hardware Support

Without access to a CXL-connected cache-coherent FPGA, we emulated the necessary hardware primitives in our evaluation. The Caching Handler emulates *cache-remote-data* by instrumenting application reads and writes to remote memory. The Dirty Data

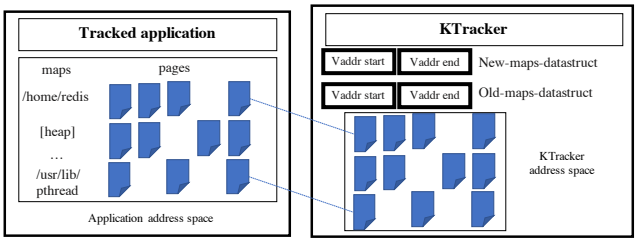


Figure 3: The KTracker simulator and its data structures.

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. 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 disaggregated memory* operation without page faults. KCacheSim measures the average memory access time (AMAT) by relying on an existing cache simulator, Cachegrind [39], to determine the cache miss rates for each application from each level of the cache. 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. 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.

(2) Measuring dirty data amplification. We developed KTracker to emulate Kona dirty data tracking at cache-line granularity by comparing snapshots of the application's memory in software (Fig. 3). 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.

4.5 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. 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. 1).

5 EVALUATION

In this section, we evaluate Kona's end-to-end performance using hardware emulation and simulation. We compare Kona with a virtual memory remote memory system using a microbenchmark (§5.1) and we use our software tools to evaluate disaggregated memory operations on real applications: fetching remote data (§5.2) and dirty data amplification (§5.3).

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].

5.1 End-to-end Performance

We compare Kona with a virtual memory-based implementation (Kona-VM) using a benchmark that we instrument to emulate disaggregated memory access. Kona-VM is a good baseline for 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 4). Kona only writes the dirty cache-lines to the remote host, while Kona-VM has to write entire pages.

Next, we evaluated the benchmark with all the initial data in disaggregated 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. Kona-VM-NoWP avoids write-protection, 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.

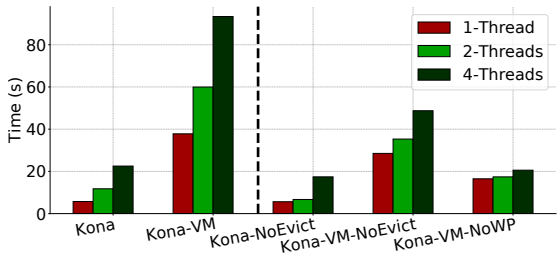
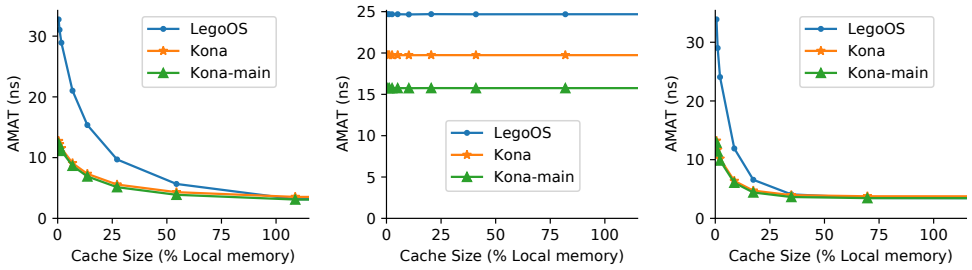


Figure 4: Kona and Kona-VM.

5.2 Fetching Remote Data

In this section, we evaluate the remote data fetch operation using KCacheSim to study the average memory access time (AMAT) for



(a) Redis Rand **(b) Linear Regression** **(c) Graph Coloring**

Figure 5: Simulating remote data fetch

applications accessing disaggregated memory. KCacheSim simulates Kona's Caching Handler 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 disaggregated 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 (via processor architecture extensions). For LegoOS, the memory hierarchy includes hardware caches, CMem (locally attached DRAM) and disaggregated memory. We measured remote access latencies running on real hardware (*not* in simulation), including page fault overheads and we use these measurements for the simulation (10 μs).

For large cache sizes, close to 100% of application peak resident set sizes, all systems perform similarly because the number of remote accesses is small. However, the AMAT increases quickly for smaller caches, as the applications incur more (expensive) remote memory accesses. As Fig. 5 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 lower AMAT than LegoOS. The one exception is Linear Regression (Fig. 5b), 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.

5.3 Dirty data amplification

We use KTracker (§4.4) to simulate cache-line tracking and evaluate dirty data amplification difference compared to 4KB page granularity. 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. We show the 4KB-page amplification *relative to cache-line tracking* in Figure 6 for Redis-Seq (sequential access) and Redis-Rand (random access). Redis-Seq 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.

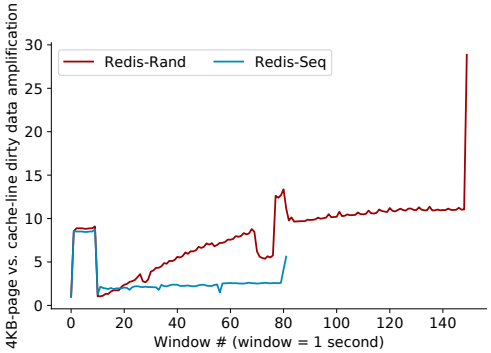


Figure 6: Reducing dirty data amplification.

6 RELATED WORK

Cache coherence ensures consistency between multiple cached copies of a memory location [60, 64]. The memory controller maintains access permissions for all cache-lines belonging to its physical memory and has visibility into CPU cache-line reads and writes.
Cache-coherent FPGAs (ccFPGAs) are connected to the CPU(s) using a link that ensures memory coherence between a CPU-attached memory and an FPGA-attached memory (e.g., CXL [74]). ccFPGAs [3, 28, 41, 51] can observe the CPU's local memory coherence events and use this information to enable new disaggregated memory systems. FPGAs have been used in datacenter [27, 65] for different purposes, such as accelerating applications [12–14, 22, 35, 38, 47, 55, 62, 73, 75, 76], smart NICs [34, 58], and multi-tenancy [46, 54, 83].
Disaggregated memory systems. Distributed shared memory [16, 19, 48, 69, 70] provides 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. Works on disk swapping [1, 6, 44] rely on page faults and page-based tracking, which limits their performance. Recent page-based disaggregated memory systems also use virtual memory [10, 15, 36, 57, 71, 81]. 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 applications.

Sub-page granularity memory access tracking. Prior works have explored sub-page granularity 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]. These approaches trade-off generality, tracking granularity, and application performance based on the specific use-case.

7 CONCLUSION

Disaggregated memory brings numerous benefits, which cannot be achieved by increasing the amount of memory on each host. Such benefits include decreasing capital and operating expenditures by improving memory utilization, allowing scaling compute and memory independently, as well as decreasing memory over-provisioning on each host and reducing the number of premium CPUs required, which come as a pre-requisite for large-memory hosts.

Despite the clear motivation for disaggregated memory, achieving high performance in disaggregated memory systems remains challenging. This paper introduces a new class of systems for disaggregated memory that uses the local host's cache coherence mechanisms to track applications' memory accesses in order to improve application performance, dirty data amplification and network utilization. Our approach requires adding a cache coherent FPGA to each host – a small cost, relative to the cost savings enabled by making disaggregated memory practical.

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.