# CaRDS: Compiler-Aided Remote Data Structures

Brian R. Tauro[*]
briantauro7@gmail.com
Illinois Tech
Chicago, Illinois, USA

Ian Dougherty[†]
ianedougherty01@gmail.com
Illinois Tech
Chicago, Illinois, USA

Kyle C. Hale
kyle.hale@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

## Abstract

Far memory tiers improve memory utilization by enabling memory intensive applications to use idle memory from other machines over the network. Recently, compiler approaches to far memory have demonstrated how static analysis can be leveraged to automatically transform applications to make efficient use of remote memory tiers. However, policies in these compilers, e.g., the determination of whether objects should be remoted, prefetched, or evacuated are made conservatively at compile time or require profiling. While profiling can alleviate conservative policies, profile-guided systems can be expensive and may not work well for applications that have variation in their inputs. We propose CaRDS, system that combines both runtime and static analysis to determine far memory policies dynamically, at data structure granularity, and without profiling. CaRDS remoting policies can outperform prior automatic approaches by up to ~2× and are within 25% of profile-guided systems when the local memory is highly constrained.

**ACM Reference Format:**
Brian R. Tauro, Ian Dougherty, and Kyle C. Hale. 2025. CaRDS: Compiler-Aided Remote Data Structures. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767488

## 1 Introduction

Software memory disaggregation is an increasingly popular technique that aims to improve memory utilization by leveraging unused memory (far memory) from neighbouring machines over fast interconnects, primarily to accommodate memory intensive applications [19, 27]. For applications with amenable memory access patterns, far memory systems can perform similar to a setup with only local memory. Contemporary software-based implementations fall into three categories: the most common repurposes the swap subsystem in the OS to fetch remote objects in response to page faults. While the developer need not modify code to run on such a system (thus achieving full transparency), the performance overheads can be significant, primarily due to frequent page faulting [3, 10, 24, 25, 30, 31]. Because kernel-based solutions lack high-level knowledge of application data structures, they must rely on complex prefetching and eviction schemes to sufficiently amortize the cost of remote transfers. Moreover, these solutions are constrained by the architected page size of the hardware, and have a higher bandwidth consumption stemming from I/O amplification [6]. Library-based systems overcome the performance penalties of kernel-based systems but trade off transparency, i.e they require source code modifications. AIFM, an exemplar of this method, requires developers to replace existing data structures with special structures provided by the runtime [25]. This enables the runtime to determine prefetching and evacuation policies for each data structure, and allows the runtime to select appropriate far memory policies. A third alternative, the compiler-based approach, requires only recompilation with a custom far memory compiler to achieve both transparency and performance. Notable examples include Mira [11] and TrackFM [26]. However, due to the limitations of static analysis, these compilers must either use profiling liberally, or make conservative decisions about which objects to remote, and when they should be prefetched and evacuated. For example, in TrackFM, all objects are assumed to be remotable, since the compiler is unable to predict locality of access statically. information at static time to make the right decisions. Mira takes an alternative approach, using profiling traces to characterize data structures' locality of access, thus informing remoting, prefetching and evacuation policies. While profiling does side-step the compiler's inability to determine runtime access patterns, it still may require several runs of the application to sufficiently capture representative traces.

In this paper, we seek to augment the compiler approach by drawing inspiration from library-based approaches [25], which can leverage user hints to identify and annotate remote data structures, enabling the runtimes to make policy decisions dynamically. However, without user hints or profiling runs, the compiler is unable to match this ability. First, source-level information (e.g., custom data structures) that might help characterize a data structure can be lost during compilation. Second, the compiler's analysis passes are limited to only static information, losing out on dynamic behavior that can inform relevant runtime policies. We overcome the first limitation by leveraging data structure analysis [13, 14] to automatically recover data structure semantics during compilation. We overcome the second limitation by co-designing the compiler and runtime, i.e., by passing compiler-identified data structure information to the runtime. This enables the runtime to make policy decisions *on a per data structure basis* dynamically and obviates overly conservative policies. This approach is fully automatic and does not require code changes.

[*]Now at Intuitive.
[†]Now at Group One Trading, LLC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*SC Workshops '25, St Louis, MO, USA*
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767488

## 2 Related Work

The most closely related compiler-based systems are Mira [11] and TrackFM [26], which lack support for recursive data structures, and require profiling or overly conservative remoting policies. Hardware-based far memory systems explore new data structures to increase efficiency. For instance, Aguilera et al. propose novel hardware primitives to improve data structure performance [1]. CXL hardware supports disaggregation at cache line granularity, enhancing memory performance [7], but such hardware is still scarce. CLIO introduces an advanced virtual memory system, custom network stack, and offloading capabilities to improve hardware handling of far memory operations [12]. Similarly, KONA brings updates to virtual memory systems, enabling support for smaller object sizes [6]. Finally, MIND demonstrates how integrating memory management logic into the network fabric can maintain shared memory coherence while enhancing performance [18]. CaRDS does not require custom hardware, so can be used with commodity, off-the-shelf hardware.

Sofware-based far memory has been implemented in both user-space and in the kernel. Kernel-based solutions swap pages to far memory over high-bandwidth RDMA networks [3, 10, 24, 30]. These systems attempt where possible to take kernel page faults off the critical path by leveraging asynchronous/offload mechanisms, and by improving the prefetching and reclaim logic in the OS. Leap [2], Canvas [28], and 3PO [5] explore prefetching and application isolation techniques at runtime. Compiler prefetching for recursive data structures has been explored for SMP systems [20, 21]. but relies on virtual memory support that does not apply to far memory.

Finally, there is a large body of work we build on that focuses on identifying data structure semantics at compile time. Existing compilers have explored data structure analysis for SMPs [14], but not in the context of disaggregated memory. SeaDSA extends and improves these methods for software verification [13]. In CaRDS we utilize SeaDSA to detect disjoint data structures and improve their performance. Pool-based allocation for disjoint data structures [16], shows the benefits of compiler-aware memory allocation for data structures in SMP systems. CaRDS draws inspiration from the pool allocation work and leverages similar ideas for far memory.

## 3 CaRDS Design

There are two fundamental challenges that limits a compiler's ability to automatically transform an existing application's data structures to use remote memory. The first challenge is the loss of information that occurs when source code is compiled down to an intermediate representation (IR), in our case LLVM IR. The LLVM type system does not recognize user-defined types, thus limiting the flow of data structure information to middle-end transformation pipelines. In particular, for a particular load/store operation in LLVM IR, there is no direct way to determine which data structure the operation corresponds to. Identifying an application data structure enables the compiler to supply the runtime with both the data structure's prototypical access patterns and its allocation sites.

Our approach overcomes this limitation by building on prior work in inter-procedural data structure analysis (DSA). DSA was originally designed to identify connectivity between memory objects across the entire program. Unlike shape analysis [29], which focuses on classifying data structures, DSA primarily concerns itself with how memory objects are connected within a program, thus improving compilation times for the analysis. In previous work, DSA has been used to capture properties of memory objects, such as determining whether they are type-safe or optimizing data structures for memory efficiency [17]. Here, we leverage DSA to automatically identify disjoint data structures, enabling their transformation into remotable data structures [13, 16]. Additionally, DSA analysis is context-sensitive, meaning it can identify specific instances of a data structure. This capability is particularly useful for our purposes, as it allows us to assign unique prefetching and remotable policies to each instance.

To effectively enforce our policies for each data structure at runtime, we need a way to link the compiler-identified data structures to the application's allocation requests and memory access patterns. For example, when we see a memory allocation request—e.g., a call to `malloc`—CaRDS must determine to which data structure the allocation corresponds, and given the characteristics of that data structure, whether or not the allocation should thus be remotable or not. Fortunately, prior work on pool allocation [16] demonstrates how disjoint memory pools can be passed to the runtime by leveraging data structure analysis. Specifically, the pool allocation technique partitions heap data structure instances into pools and passes pool handles to the runtime, allowing for memory optimizations tailored to the data structure instances. In CaRDS, we implement pool allocation to establish far memory policies for each data structure.

The second challenge is that the allocation size of data structures are not always known at compile time. In CaRDS, we develop policies that are co-designed with the runtime to address the lack of runtime information at compile time. For example, the policy for determining whether a memory allocation should be remotable is evaluated at runtime in CaRDS. This is done by using compiler-provided data structure information to identify the data structure responsible for the allocation, while also considering runtime factors such as the memory usage on the system.

Figure 1 depicts the high-level architecture of CaRDS, which is built atop TrackFM. Developers need only recompile their application using our compiler framework to run their legacy application using far memory.

```c
int * ds1, * ds2;
double * alloc() {
  return malloc(ARRAY_SIZE);
}
void main() {
  ds1 = alloc();
  ds2 = alloc();
  Set(ds1, 0);
  Set(ds2, 1);
  for (int k=0; k<NTIMES; k++)
    Set(ds2, k);
}
void Set(int * ds, int val) {
  for (j=0; j<ARRAY_SIZE; j++)
  ds[j] = val;
}
```

**Listing 1: Example C code with two data structures.**

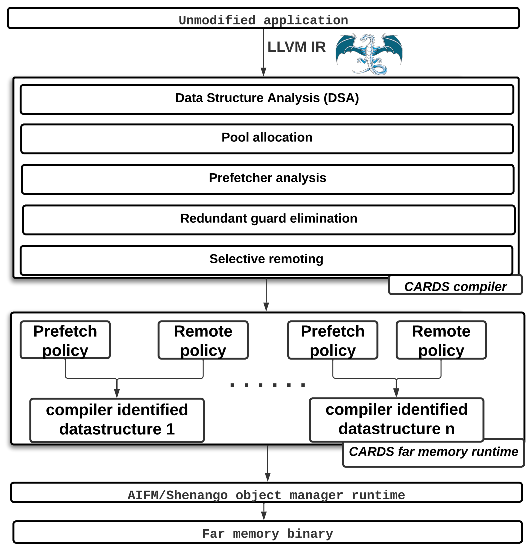


**Figure 1: High-level overview of CaRDS, which automatically transforms an application to far memory using both compile-time and run-time techniques.**

## 4 Implementation

We first outline the compiler passes in CaRDS. We then describe CaRDS's remoting and prefetching policies, which do not require code modifications or profiling runs.

Listing 1 depicts two data structures (`ds1`, `ds2`) that being initialized, with `ds2` specifically being initialized within a loop. We use this example to illustrate how the CaRDS compiler and runtime information are combined to automatically transform data structures and make appropriate policy decisions, as discussed in the following sections.

### 4.1 Compiler optimizations

CaRDS builds on NOELLE to construct its compiler passes [23]. NOELLE offers high-level, program-wide abstractions in turn built LLVM IR [15].

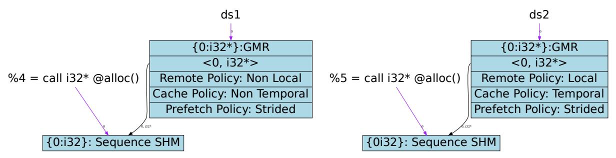


**Figure 2: DSA analysis for Listing 1**

*DSA Pass.* CaRDS leverages SeaDSA [13] to detect disjoint data structures. This is an inter-procedural and context-sensitive analysis, enabling it to capture more data structures than the prior DSA approach [16]. Figure 2 shows the disjoint data structures identified for Listing 1 by CaRDS. Notably, only heap-allocated data structures are identified.

**Algorithm 1** Lattner and Adve's pool allocation algorithm [16].

```
1:  for all $F \in$ functions($P$) do
2:      seadsa::graph G = DSAGraphForFunction(F)
3:      for all $n \in$ Nodes($G$) do
4:          if ($n.alloc == Heap\ and\ escapes(n)$) then
5:              $ds\_handle \leftarrow AddDSHandleArg(F, n)$
6:              $dsmap(n) \leftarrow ds\_handle$
7:              $argnodes(F) \leftarrow$ argnodes(F) $U$ $\{n\}$
8:          else// Node is local to fn
9:              $ds\_handle \leftarrow DS\_INIT(F, n)$
10:             $dsmap(n) \leftarrow ds\_handle$
11:         end if
12:     end for
13: end for
14: for all $F \in$ functions($P$) do
15:     for all $I \in$ instructions($F$) do
16:         if $I\ isa\ malloc\ callinst$ then
17:             replace I with dsalloc(size,dsmap(N(ptr)))
18:         else if $I\ isa\ callinst$ then
19:             for all $n \in$ ArgNodes($Callee$) do
20:                 $addCallArg(dsmap(NodeInCaller(F, I, n)))$
21:             end for
22:         end if
23:     end for
24: end for
```

*Pool Allocation.* After identifying disjoint data structures, CaRDS conveys the extracted information to the runtime by utilizing the pool allocation algorithm shown in Algorithm 1. Since the original implementation of this algorithm was not actively maintained with newer LLVM versions, we reimplemented it for our purposes in the NOELLE framework. Unlike the original algorithm, which employs bottom-up, inter-procedural analysis to identify disjoint data structures, CaRDS uses context-sensitive disjoint data structures identified by our DSA analysis and passes them to the pool allocation algorithm 1.

```
double * alloc(int DH) {
  return cards_malloc(ARRAY_SIZE, DH);
}
void main() {
   //ds_init(ds_id, cache_policy, prefetch_policy,
  remote_policy)
   int dh1 = ds_init(1, false, STRIDED, REMOTABLE);
   int dh2 = ds_init(2, true, STRIDED, LOCALIZE);
  ds1 = alloc(dh1);
  ds2 = alloc(dh2);
}
```

**Listing 2: CaRDS pool allocation transformation for Listing 1**

The algorithm is used to initialize data structures and link memory allocations sites to their corresponding data structures. It works in two phases. In the first phase (lines 1–13), the algorithm modifies functions in the program that might allocate memory on the heap. This includes both direct calls to `malloc` and indirect calls that eventually lead to `malloc` The algorithm uses a map, `dsmap`, to track a handle for each data structure. If a function returns a pointer or passes it outside the function (rendering it an *escaping pointer*), the algorithm adds extra arguments to the function to handle the data structure properly. If the pointer does not escape, the algorithm inserts a call into the CaRDS runtime to initialize the data structure, and the handle is saved in dsmap. In the second phase (lines 14–24), the algorithm updates the functions to include the data structure handles from dsmap as arguments. This ensures that any function that calls the modified functions from the first phase knows which data structure it is working with, so memory can be managed correctly at runtime. For more details on pool allocation, readers are referred to the original paper [16].

Listing 2 illustrates how CaRDS pool allocation transforms Listing 1 and employs pool allocation to segregate data structures with different prefetch and remoting policies. Each data structure is assigned a unique handle, DH, that is appended to the non-canonical bits of a pointer, facilitating the mapping of pointer addresses to their corresponding data structures.

*Prefetching analysis.* In this pass, we gather application type information in the LLVM IR and utilize induction variable analysis to identify sequential access patterns. The CaRDS compiler analysis operates at the level of individual data structures, with each data structure assigned its own prefetching policy. Figure 2 highlights the data structures with strided access patterns identified at compile time for Listing 1.

*Redundant guard elimination.* In a far memory system, memory accesses to data structure objects can take place either when the objects are located in local memory or in remote memory. In line with previous far memory compiler systems, CaRDS automatically inserts guard checks on memory instructions to ensure proper localization of objects (safety). Unlike TrackFM's optimizations, which only apply to induction variables, CaRDS guard optimizations apply to non-induction variables as well, allowing it to work with more complex data structures.

```
void Set(int * ds, int val) {
  remote = check_remotable_policy(ds); // loop v1
  if (remote) {
    for (j=0; j<ARRAY_SIZE; j++) {
      if (safe_to_access(&ds[j]))
        ds[j] = val;
    }
  } else {
    for (j=0; j<ARRAY\_SIZE; j++) // loop v2
      ds[j] = val;
  }
}
```

**Listing 3: CaRDS selective remoting transformation for Listing 1.**

*Selective remoting.* The advantage of treating some data structures as non-remotable is that accesses to them (perhaps because they exhibit very little spatial or temporal locality) can elide guard overheads. However, this elision is not always feasible at compile time due to a lack of runtime information. Since the memory access pattern associated with a data structure is unknown at compile time, it is in general not possible to determine whether the data structure should be marked as remotable at compile time. This limitation

**Table 1: Comparison of primitive overheads for CaRDS and TrackFM. Costs are reported in median cycles over 100 trials.**

| Runtime Event | Local Cost | Remote Cost |
|---|---|---|
| CaRDS read fault | 378 | 59K |
| CaRDS write fault | 384 | 59K |
| TrackFM read guard | 462 | 46K |
| TrackFM write guard | 579 | 47K |

can be mitigated through profiling or using techniques like loop peeling, where certain iterations of a loop are peeled out. These peeled iterations can be used at runtime to help decide whether the larger loop should be remotable. However, determining the number of profiling runs or loop iterations needed to make an accurate decision is not always straightforward. Instead, in CaRDS, we use code versioning by maintaining two versions of the code: one that is instrumented and another that is not. The uninstrumented version is selected only if all data structures are deemed localized, ensuring that unnecessary instrumentation is avoided.

Listing 3 demonstrates our transformation applied to Listing 1. Before executing a loop, the CaRDS compiler injects a call into the runtime to check whether the data structures used within the loop are remotable. If all data structures are marked as non-remotable, the execution branches to the uninstrumented version.

## 4.2 CaRDS runtime system

CaRDS utilizes a modified version of the AIFM runtime to manage far memory objects at the granularity of data structures. During compile time, CaRDS identifies disjoint data structures and assigns each a unique data structure ID, which is employed at runtime, as illustrated in Listing 2. The data structure ID is appended to the non-canonical bits of pointer addresses during memory allocation calls to manage these associations. CaRDS monitors cache hits and misses for each memory object, leveraging these statistics on a per-data structure basis to inform runtime policy decisions. Unlike previous compiler-based far memory systems, CaRDS allows policy decisions such as prefetching and remotability at the individual data structure level, offering finer control over memory management.

```
0: shr    $0x30,%rcx          // custody check (is this CARDS-managed ptr?)
1: je     4                   // if not, perform original load/store
2: callq  <card_deref_fn> // otherwise, runtime call
4: mov    %rbx,(%rax)         // TARGET LOAD/STORE
```

**Figure 3: CaRDS guard lowered to x64 code.**

*CaRDS guards.* CaRDS manages far memory at the object level, allowing for objects of arbitrary sizes to reside in either local or remote memory. To ensure memory safety, it is essential to localize an object before access; CaRDS does this by injecting guards on accesses to objects.

In CaRDS, an object may map to multiple addresses, determined by its size. The size of an object is guided by compiler hints provided to the runtime during the data structure initialization (ds_init).

```
uint64_t cards_deref(uint64_t addr) {
  //get ds handle from non canonical bits
  uint64_t ds_id =  (addr >> ORT_POS);

  DS * ds = ds_list[ds_id];

  //map address to object
  uint64_t ind = off >> ds->obj_shift;

  FarMemPtr * obj = ds->pool_manager->ptrs_[ind];

  //if object already in local memory
  if (safe_to_access(obj))
    return obj.paddr;

  //fetches object over the network
  LocalizeObject(ds, ind, obj);

  return obj.paddr;
}
```

**Listing 4: CaRDS deref function.**

For instance, a declaration like `char ds[4096]` could correspond to a single CaRDS object if the object size for the data structure is set to 4K. Consequently, CaRDS data structures can have varying object sizes based on the static hints given by the compiler.

Figure 3 illustrates a CaRDS guard check. If a memory address has its non-canonical bits (bits 48-63) set (known as a custody check), CaRDS injects a call to the `cards_deref function`. This function is responsible for ensuring memory safety. It first maps the higher-order address bits to their corresponding data structure and then uses the lower address bits to associate with the actual object. Following this mapping phase, the system checks whether an object is localized; if it is not, CaRDS calls into the AIFM runtime to fetch the object. CaRDS injects guard checks once for every object access within a loop. If multiple memory locations map to the same object, a check occurs only once, thanks to the redundant guard optimization described above.

*Remoting policy selection.* CaRDS manages local memory by dividing it into two categories: pinned memory, which cannot be remoted, and remotable memory, from which data structures marked as remotable are allocated. The system uses a custom libc to control memory allocations, associating each allocation with a specific data structure. During an allocation, CaRDS evaluates static information about the data structure provided by the compiler to determine if the memory should be allocated from remotable memory.

The primary challenge in determining whether a data structure should be non-remotable is that the size of the data structure may not be known at compile time, which complicates the decision on which data structures should be localized. Previous compiler approaches have relied on profiling to first determine the sizes of memory objects, and then, in subsequent runs of the program, decide whether the object should be remotable. For example, in Mira [11], a memory profiler is used to determine allocation sizes, and only those objects with large sizes are further analyzed to decide on the appropriate far memory policies. In contrast, CaRDS addresses this challenge by developing policies that are independent of data structure size. Rather than relying on memory size, CaRDS uses a tunable parameter which determines the percentage of data structures that should use non-remotable memory. Ideally, this parameter is set higher when more local memory is available and lower when memory is limited, offering a more dynamic and flexible approach to memory management.

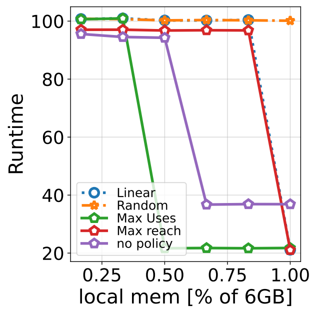


**Figure 4: CaRDS performance across different remotable policies for Listing 1 when top k=50% of data structures are marked as non-remotable.**

Due to limited runtime information, it is sometimes impractical to make accurate decisions about remoting at compile time. To address this, the CaRDS runtime can override static hints as needed. For example, if a data structure does not fit in local memory despite a static hint for localization, the runtime may choose to remote it. In cases where dynamic data structures grow during execution, the runtime tracks allocations to ensure they remain local. This allows certain code paths to execute without instrumentation for non-remotable data structures. When a data structure is remoted, CaRDS ensures use of the instrumented code path.

Below, we outline several policies that can enhance the selection of data structures to be localized based on static code information:

*Linear Assignment.* This policy allocates pinned memory (non-remotable) sequentially in program order, switching to remotable memory once local memory is exhausted.

*Random Assignment.* This policy allocates pinned memory randomly for memory allocations.

*Maximum Reach.* This policy prioritizes data structures used in functions with long caller/callee chains, marking them as local (non-remotable) at compile time. CaRDS uses the strongly connected components (SCC) call graph to identify which functions access a particular data structure. When pinned memory is fully utilized, CaRDS defaults to remotable memory. The data structures used in the top k functions with long caller/callee chains are marked as non-remotable.

*Maximum Use.* This policy prioritizes data structures with high memory usage as non-remotable candidates. The top k data structures are marked as non-remotable, ensuring that frequently used data structures remain local.

$$ds = MAX(\#loops + \#functions) \quad (1)$$

In Listing 1, we observe that `ds2` has a higher usage than `ds1` and benefits more from localization. By implementing the policy that prioritizes data structure usage, CaRDS can make more informed decisions; using the linear policy may lead to suboptimal choices.

In Figure 4, we compare various policies for Listing 1, allowing one of the data structures to be localized based on compiler policy by setting the threshold parameter k = 50%. Both data structures in Listing 1 are allocated 3 GB of memory. When 50% of local memory is available, one of the data structures can be localized. A naïve policy would localize ds1, while a refined policy correctly localizes ds2, resulting in improved performance by minimizing network communication, as shown in Figure 4.

*Prefetching Policy Selection.* A variety of prefetching policies are available, each differing in complexity, specificity, and aggressiveness. For CaRDS, we have chosen to support existing compiler prefetchers, including a majority stride-based prefetcher, a greedy recursive prefetcher, and a jump pointer prefetcher [22]. Based on the static and dynamic information available for each data structure, CaRDS selects the most appropriate prefetch policy. Standard prefetching metrics, such as accuracy and coverage, are used to evaluate the effectiveness of each prefetching policy. The combination of static and dynamic information per data structure creates opportunities for advancing prefetching algorithms in CaRDS.

## 5 Preliminary Evaluation

CaRDS combines static and runtime information to determine optimal far-memory policies for each data structure. Our evaluation assesses the performance impact and advantages of combining compiler and runtime optimizations at the granularity of data structures, enabling informed decisions on remoting and prefetching policies. We then demonstrate that CaRDS serves as a viable, non-profiling alternative, outperforming conservative methods while maintaining acceptable performance relative to profiling-based approaches. Our evaluation aims to address the following questions:

- How does CaRDS's remoting policies benefit applications? (§5.1)
- How does CaRDS's prefetching policies benefit applications? (§5.2)
- How does CaRDS perform with realistic applications? (§5.3)

**Experimental setup:** Our experiments were conducted on CloudLab [8] with two x170 machines, each with 10-core Intel Xeon E5-2640v4 2.4 GHz CPUs. These machines have 64GB RAM and a 25 Gb/s Mellanox ConnectX-4 NIC. We used Ubuntu 18.04, Linux kernel version 5, and DPDK version 18.11 (also used by AIFM). CaRDS builds on LLVM version 14.0.6, [1] with NOELLE v14.1.0.[2]

*Application benchmarks.* We select three benchmarks to evaluate CaRDS, among which the analytics benchmark and breadth-first search (BFS) represent common access patterns in the wild.

*analytics workload* is a data analytics application that uses the 2014 NYC taxi trip dataset from Kaggle[3] to analyze New York City taxi trips. We choose this benchmark to compare against existing far-memory compilers (both profiling and conservative) and validate our results against TrackFM [26]. For Mira, we were unable to reproduce the NYC benchmark due to the incomplete Mira implementation[4]; instead, we use a projected curve based on the results from their paper [11]. The memory working set size for the taxi-trip workload was 31 GB, and the dataset size was 16 GB.

Next, we choose a widely used benchmark suite, PolyBench, which is a collection of benchmarks with static control parts [9]. Within PolyBench, we select *ftfdapml* (Finite Difference Time Domain Kernel using Anisotropic Perfectly Matched Layer), which is used to simulate optical devices and model electromagnetic wave interactions with materials. We select ftdapml because it has the largest number of data structures in the PolyBench suite, making it useful for evaluating remoting policies in CaRDS. The memory working set size of ftdapml is 8 GB.

Finally, we select *BFS*, a graph workload commonly used in datacenters. BFS is characterized by an irregular access pattern. We use the BFS benchmark from the GAPS benchmark suite [4]. The memory working set size for BFS is 1.2 GB.

### 5.1 Evaluation of CaRDS remoting policy

In far memory systems, ideally, hot data structures should not be remotable (i.e., they should use pinned local memory) to avoid the cost of network fetches. However, marking a data structure remotable becomes necessary if their memory requirements exceed a single server's capacity. In this section, we evaluate CaRDS's remoting policies, which are determined dynamically without profiling or conservative assumptions, and compare them with existing profiling and conservative compilers.

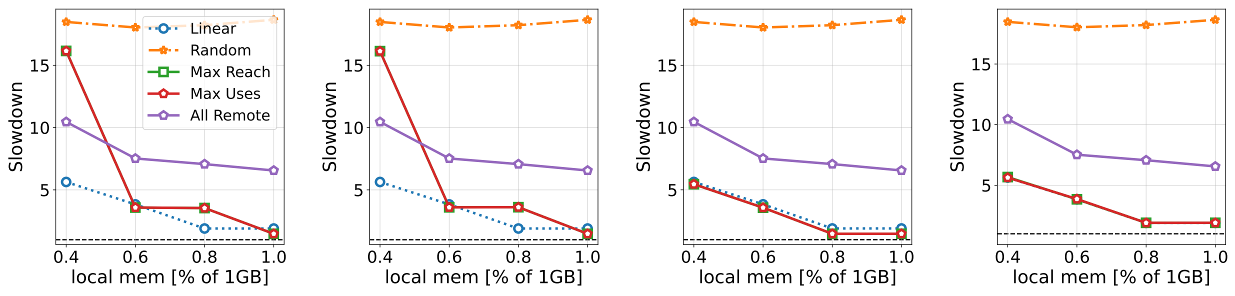
Figures 5 to 7 compares the remoting policies of CaRDS (as discussed in Section 4) across three application benchmarks. As local memory increases, more data structures are designated as non-remotable, meaning their memory allocations are pinned to local (non-remotable) memory. These figures show that selectively remoting data structures can improve application performance by up to ~2×, especially when sufficient local memory is available.

For the analytics workload, CaRDS allocates available local memory by setting aside 1 GB for remotable memory, while the remaining local memory is used for pinned memory, if available. CaRDS identifies 22 disjoint data structures at compile time, each of which is assigned a dedicated prefetcher and a custom remote policy manager. Since the sizes of these data structures are not known at compile time, CaRDS includes a tunable parameter, k, that controls the percentage of data structures to be localized.

When all data structures are localized (k = 100), the policies exhibit similar performance, except for the random policy. However, as the number of localized data structures decreases, the effectiveness of each policy varies. The "Max Reach" policy proves to be more resilient to selective remoting. In scenarios where the application's local memory exceeds 90%, a linear policy suffices because all data structures can be localized on demand. Unlike other policies that statically assign certain data structures to be remotable, the linear policy makes decisions at runtime.

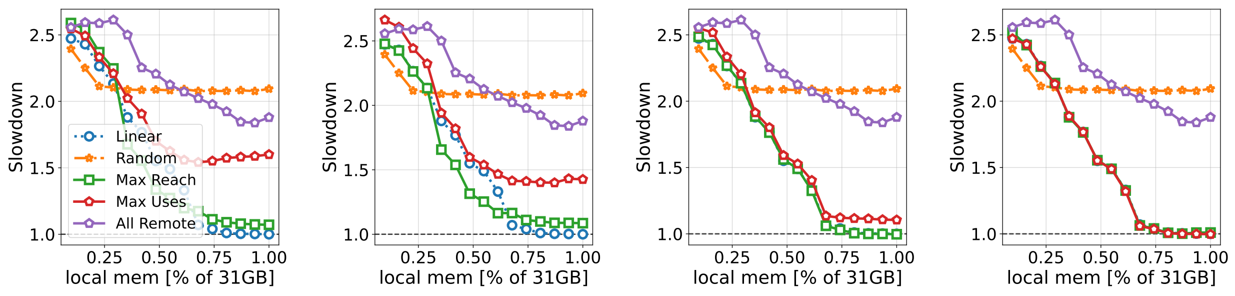
In Figure 7 we allocate 1 GB of remotable memory for the ftfdapml benchmark, with the remainder designated for pinned (non-remotable) memory. CaRDS identifies 15 disjoint data structures at compile time, with certain data structures requiring more memory than others. The "Max Use" policy performs better when k exceeds 25%. Additionally, both the "Linear" and "Max Reach" policies

[1]commit f28c006
[2]commit 68f334a
[3]kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook
[4]https://github.com/WukLab/Mira



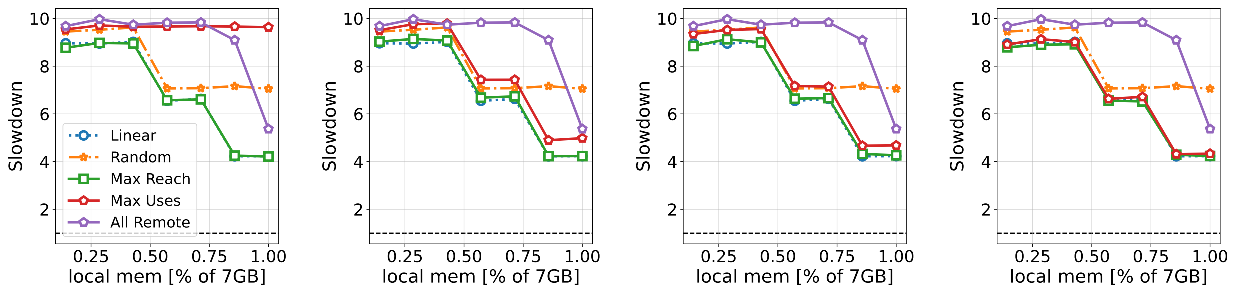
(a) top k = 25% (b) top k = 50% (c) top k = 75% (d) top k = 100%

**Figure 5: CaRDS remotable policies for BFS where we increase the amount of data structures to be localized from left to right. The linear policy is unaffected even when the top 25% of data structures are localized.**



(a) top k = 25% (b) top k = 50% (c) top k = 75% (d) top k = 100%

**Figure 6: CaRDS remoting policies for the analytics workload, where we increase the amount of data structures to be localized from left to right. The "Max Reach" policy is unaffected even when data structures are localized only in top 25% of functions.**



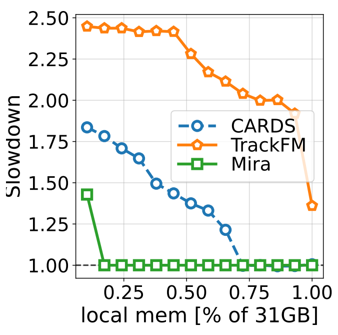
(a) top k = 25% (b) top k = 50% (c) top k = 75% (d) top k = 100%

**Figure 7: CaRDS remoting policies for ftfdapml where we increase the amount of data structures to be localized from left to right. The "Max Reach" policy is unaffected even when data structures are localized only in top 25% of functions.**

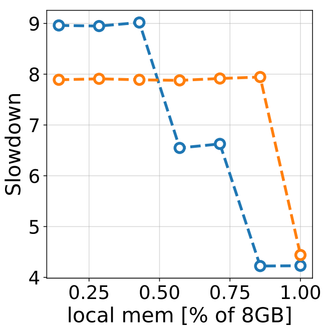
demonstrate greater tolerance to selection changes, achieving performance that is approximately ~4× better than the all-remotable configuration.

In Figure 5, the BFS benchmark is allocated 256 MB for remotable memory, with the rest designated for pinned memory. CaRDS identifies 19 disjoint data structures, with varying memory requirements among them. The "Linear" policy consistently outperforms other policies across different selections of data structures.

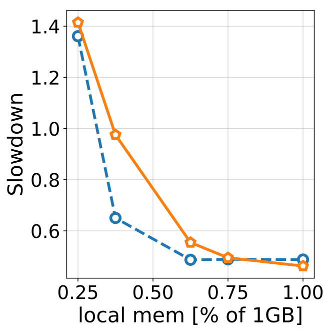
When all data structures are marked as remotable, approximately 10 billion guard checks are performed across the three benchmarks, which can become prohibitively expensive when sufficient local memory is available. Importantly, these policies not only eliminate guard checks but also reduce network communication.



**(a) analytics workload**



**(b) ftfdapml**



**(c) BFS**

**Figure 8: CaRDS performance compared to prior far memory compilers. CaRDS is within 20% of Mira when local memory is less than 25%, and outperforms TrackFM consistently when more than a few data structures are localized.**

In summary, Figures 5 to 7 demonstrate that, with the exception of the random policy, all other policies enhance the efficiency of far memory systems compared to the conservative approach of marking *all* data structures as remotable candidates. When objects are conservatively designated as remotable at compile time, they can lead to performance overheads in both scenarios: when local memory is limited and when it is adequate. Specifically, if local memory is constrained and all data structures are marked as remotable, the system incurs frequent network requests to retrieve objects. On the other hand, when local memory is plentiful, the system suffers from the overhead associated with static guard checks. In both situations, marking all objects as remotable can result in considerable network and instrumentation overheads. Thus, selectively remoting data structures using even simple policies like linear assignment proves to be more effective than overly conservative approaches.

## 5.2 Evaluation of CaRDS prefetch policy

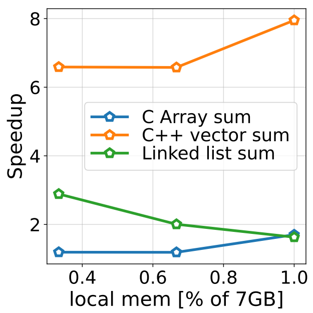


**Figure 9: CaRDS speedup compared to TrackFM for pointer chasing data structures. As CaRDS can identify disjoint data structures that have their own prefetchers, CaRDS outperforms TrackFM consistently.**

CaRDS identifies disjoint data structures within an application, with each structure having its own dedicated prefetcher. We assess the advantages of this separation by comparing CaRDS to TrackFM. In Figure 9, we analyze various data structures that perform the sum operation (`c[i] = a[i] + b[i]`). The memory working set size is set to 7 GB, and we measure the speedup of CaRDS relative to TrackFM. Our findings show that data structures with easily discernible induction variables, such as array sums, run efficiently om TrackFM. However, for pointer-chasing benchmarks like C++ vectors and maps, CaRDS consistently outperforms TrackFM since TrackFM relies only on induction variables for prefetching.

## 5.3 Application study

Figure 8 compares the performance of CaRDS with prior far memory compilers. We observe that CaRDS achieves performance that falls between profiling and non-profiling compilers for the analytics workload. Notably, CaRDS consistently outperforms TrackFM by up to approximately 2×, especially when more than 50% of the local memory is available. With 25% of non-remotable memory, CaRDS is within 20% of the performance of Mira (profile-based compiler). However, as the available local memory increases, Mira surpasses CaRDS. We aim to explore improved policies to close this gap further in future work. For the other benchmarks, CaRDS consistently outperforms TrackFM.

## 6 Conclusions

We present CaRDS, a prototype compiler-assisted far memory system that automatically transforms data structures into remotable ones. Through experiments, we have demonstrated that efficient remoting and prefetching policies can be determined dynamically, without profiling, by integrating runtime and static information at the data structure level. CaRDS outperforms previous non-profiling compiler systems by up to ~2× and comes within 25% of profiling-based compilers when local memory is constrained.

## Acknowledgments

This work was made possible with support from the United States National Science Foundation (NSF) via grants CNS-2533773, CNS-2239757, and CNS-2310422. This work used resources from CloudLab [8], supported by NSF's NSFCloud program.

## References

[1] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 120–126. https://doi.org/10.1145/3317550.3321433

[2] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, USA, Article 58, 15 pages.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the* 15*th European Conference on Computer Systems* (Herakleion, Crete, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[4] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC] https://arxiv.org/abs/1508.03619

[5] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2022. 3PO: Programmed Far-Memory Prefetching for Oblivious Applications. *arXiv e-prints*, Article arXiv:2207.07688 (July 2022), arXiv:2207.07688 pages. arXiv:2207.07688 [cs.OS]

[6] 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* 26*th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3445814.3446713

[7] cxl 2024. Compute Express Link (CXL). https://computeexpresslink.org/. Accessed: 2024-1-11.

[8] 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. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1–14.

[9] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339595

[10] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the* 14*th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. USENIX Association, Boston, MA, 649–667.

[11] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 692–708. https://doi.org/10.1145/3600006.3613157

[12] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the* 27*th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3503222.3507762

[13] Arie Gurfinkel and Jorge A. Navas. 2017. A Context-Sensitive Memory Model for Verification of C/C++ Programs. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 148–168.

[14] Chris Lattner and Vikram Adve. 2003. *Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis*. Tech. Report UIUCDCS-R-2003-2340. Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society.

[16] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. Chigago, Illinois.

[17] Chris Lattner and Vikram S. Adve. 2005. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance* (Chicago, Illinois) *(MSP '05)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/1111583.1111587

[18] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561

[19] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Proceedings of the IEEE International Conference on Big Data* (Boston, MA, USA) *(Big Data '17)*. IEEE, 2884–2892. https://doi.org/10.1109/BigData.2017.8258257

[20] Chi-Keung Luk and T.C. Mowry. 1999. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Trans. Comput.* 48, 2 (1999), 134–141. https://doi.org/10.1109/12.752654

[21] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/237090.237190

[22] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-based Prefetching for Recursive Data Structures. In *Proceedings of the* 7*th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/237090.237190

[23] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, et al. 2022. NOELLE Offers Empowering LLVM Extensions. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization* (Seoul, Republic of Korea) *(CGO '22)*. IEEE, 179–192. https://doi.org/10.1109/CGO53902.2022.9741276

[24] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 181–198. https://www.usenix.org/conference/nsdi23/presentation/qiao

[25] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the* 14*th USENIX Symposium on Operating Systems Design and Implementation* (Virtual, USA) *(OSDI '20)*. USENIX Association, Berkeley, CA, USA, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[26] Tauro, Brian R. and Suchy, Brian and Campanoni, Simone and Dinda, Peter, and Hale, Kyle C. 2024. TrackFM: Far-out Compiler Support for a Far Memory World. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (to appear)* (San Diego, CA, USA) *(ASPLOS '24)*.

[27] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Proceedings of the* 15*th European Conference on Computer Systems* (Herakleion, Crete, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[28] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 161–179. https://www.usenix.org/conference/nsdi23/presentation/wang-chenxi

[29] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. 2000. Shape Analysis. In *Proceedings of the 9th International Conference on Compiler Construction (CC '00)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

[30] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. 2021. DiLOS: Adding Performance to Paging-Based Memory Disaggregation. In *Proceedings of the* 12*th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21)*. Association for Computing Machinery, New York, NY, USA, 70–78.

[31] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3552326.3567488