CachedArrays: Optimizing Data Movement for Heterogeneous Memory Systems

Mark Hildebrand
University of California, Davis
Email: hildebrandmw@gmail.com

Jason Lowe-Power
University of California, Davis
Email: jlowepower@ucdavis.edu

Venkatesh Akella
University of California, Davis
Email: akella@ucdavis.edu

***Abstract*—We propose a new framework called *CachedArrays* and a set of APIs to address the data tiering problem in large scale heterogeneous and disaggregated memory systems. The proposed framework operates at a variable size object granularity and allows the programmer to specify semantic hints about future use of data via a Policy API, which are used by a Data Manager to choose when and where to place a particular data object using a data management API, thus bridging the semantic gap between the programmer and the platform-specific hardware details, and optimizing overall performance. We evaluate the proposed framework on a real hardware platform with terabytes of memory consisting of NVRAM and DRAM on large scale ML training workloads such CNNs that exhibit different data access and usage patterns. We show that *CachedArrays* outperforms hardware caches, and can exploit many of the algorithmic-specific optimizations of prior works.**

I. Introduction

From the dawn of computing, memory has been a significant bottleneck in our quest to improve performance. An idea memory subsystem provides high bandwidth, low latency, low cost per bit, and high-capacity. However, these competing demands cannot be satisfied with a single memory technology. With the advent of emerging interconnect standards like Compute Express Link (CXL) [1], system architects are looking to satisfy these conflicting requirements by creating a memory subsystem using multiple different technologies. Heterogeneous memory naturally introduces the need for data tiering [2], [3] or moving the data that is being accessed by a program to a memory pool that suits the nature of the memory access pattern. Traditionally hardware-managed multilevel caches and operating system (OS) page migration methods have been used to address this problem. However, for emerging ML workloads with working sets in the range of terabytes and complex, application-specific data use patterns, these techniques are not effective.

Transparent hardware-managed caching can cause performance degradation when the memory sizes are hundreds of gigabytes or more [4], [5]. Implementations of hardware-managed DRAM caches such as Intel's Cascade Lake systems are inefficient due to cache-line-level metadata tracking and write amplification, which results in poor bandwidth utilization that is particularly detrimental for heterogeneous systems using phase change memory (PCM) based technologies such as Intel's Optane DC [6]. Moreover, these systems can benefit from bespoke optimizations to make data movement more efficient [6]. For example, SAGE [7] proposes new data structure design and new algorithms for graph computations, AutoTM [8] uses static analysis using ILP-based techniques to optimize data movement for a certain class of ML applications such as Convolutional Neural Networks (CNNs) For GPU systems, vDNN [9] and its derivatives [10], [11], [12], [13] exploit the unique characteristics of the backprop algorithm to augment limited GPU memory with memory on the host.

These solutions are often ad hoc for a particular application, and they often require significant rewrite of the applications. To efficiently use memory and move data in a disaggregated system, we need a framework which is easily modified for new applications, algorithms, and platforms that defines exactly what the programmer should specify and what the underlying hardware can do. We believe such a framework requires a *separation of concerns* between the application's data access, the policy used to direct the data movement, and the underlying data movement mechanism. Thus, in this paper we present *CachedArrays* as an example framework which enables mostly automatic application hints for future data with a customizable data movement policy, and an independent data manager to handle data movement between different memories. Furthermore, we believe that often one-size-does-not-fit all, and the framework should operate at a program-specific level of granularity (as opposed to the 64-byte cache block) so that it is easier for the programmer to convey semantic information and more efficient in terms of metadata tracking. This renders the overall framework more transparent to the programmer (more like hardware-managed caches), and shields the programmer from low-level details of data movement, mitigating the need for a significant application rewrite while still realizing performance benefits.

We propose the design, implementation, and evaluation of a framework called *CachedArrays* (see Figure 1) that realizes these requirements. As described in Section III, *CachedArrays* the architectural abstractions of a policy and a data manager. The policy is constructed using the data management API and using the semantic intent by the programmer conveyed to via a policy API. We describe the requirements for a data movement framework and the details of software architecture of *CachedArrays* and an API suitable for implementing large scale deep neural networks in Section III. In Section IV we discuss the details of the implementation of *CachedArrays* in Julia programming language. Since disaggregated memory

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00055

TABLE I: Summary of related work in data management in heterogeneous memory systems.

Work	Abstraction Layer	Granularity	Programmability	Mechanism
Sage [7]	Algorithm	Data Structure	Application Specific	Manual Partitioning
vDNN [9], ZeRO-Offload [13]	Application	Tensor	Application Specific	Manual Partitioning
AutoTM [8], Sentinel [3],[14], DLRM [15]	Compiler	Tensor	Transparent	Profile Guided Optimization
Nimble [16], KLOC [17], Thermostat [18], [19],[20], [21], [22],[23],[24],[25],[26]	Operating System	Page	Transparent	Virtual Memory
Memory Mode/2LM (in Intel NVRAM)	Hardware	Cache Block	Transparent	HW Managed Cache
Kona [27], täkō [28]	Hardware	Cache Block	Transparent	SW Runtime based on Cache Coherence
CachedArrays (This work)	Application	Variable Sized Object	Transparent (with annotations)	Type System/Runtime

systems with CXL are currently nascent, we instead evaluate the proposed framework on heterogeneous memory machine with both Optane Non-Volatile Random Access Memory (NVRAM) and DRAM-based memories. We show that for training Convolutional Neural Networks (CNNs) *CachedArrays* improves performance 1.4× to 2.03× over baseline hardware managed cache (called 2LM) on Intel's platform. Interestingly, we show that some of the proposed optimizations (such as eager memory freeing) can improve the performance of the hardware managed caches as well.

The main contributions of this work are as follows:

- We show that by separating the concerns of communicating the application's future data use pattern (hints/annotations), when and where to move data (policy), and the underlying data movement mechanism (data manager) leads to a performant and generic data management system.
- We present *CachedArrays*, a concrete implementation of these ideas in Julia, enabling applications with array-like data structures to transparently take advantage of heterogeneous memory with optional data usage hints. Detailed evaluation of using *CachedArrays* on a **real** heterogeneous memory (NVRAM+DRAM) platform on training large deep learning models.

II. Related Work

With the advent of heterogeneous memory platforms and the growing memory footprint of ML workloads there has been significant interest in understanding and optimizing data movement [12], [29], [30], [31], [19], [32] at all levels across the software/hardware stack. Table I summarizes works most closely related to this paper.

Some prior works have proposed data structures and algorithms specifically for DRAM/NVRAM memory systems to overcome this challenge and show significant performance improvements using their bespoke solutions [33], [8], [34], [35], [36]. In our work, we look at the optimizations these prior works have used and develop a general approach which can be applied to many different applications.

There have been many works focused on optimizing deep learning workloads in the CPU/GPU environments to overcome the memory bottleneck issue for GPUs. vDNN [9], ZeRO-Offload [13] and other related works in this line [11], [12], [10], [37] exploited the algorithmic characteristics of these workloads, such as backpropagation, to decide on data placement and movement. These solutions call for deep algorithmic changes, requiring the programmer to manually determine the reuse pattern and figure out a suitable data movement scheme to actually change the algorithm, accordingly. Other research works tried to take such decisions at the compiler level including AutoTM [8] which used ILP to optimize data movement and Sentinel [3] which used runtime profiling information. However, these solutions suffer from scalability and generalization if the workloads' reuse patterns are sparse or less straightforward to detect. Examples of such workloads include Deep-Learning Recommendation Models (DLRMs) which have sparsely accessed embedding tables. Hildebrand et al. [15] show that DLRM workloads with sparse accesses also benefit from software data movement policies, but the policy must be flexible enough to adapt to the workload. The *CachedArrays* framework addresses these shortcomings by minimizing the required algorithmic changes by programmers and enabling a specialized placement and movement policy based on characteristics of the program and the reuse pattern.

Many works proposed intelligent data migration between slow and fast memories in page granularity to perform data tiering at the OS level, using runtime characteristics such as reuse pattern and page hotness [22], [2], [38], [21], [23]. However, like hardware-based techniques, these works do not take into account future information about the data use and the semantic information from the application.

Finally, other works considered hardware/software redesign for data management in memory systems including Kona which removes the overhead of conventional virtual memory from the critical path of tracking applications [27] and täkō which enables application to be informed of the memory access behavior and customize actions based on callbacks [28]. Both Kona and täkō provide hardware and software components for better data management at the runtime. However, none of them provide support for semantic information sharing from application to the hardware, to form a suitable data management policy. Täkō overlaps with what we introduce as the data manager in Section III-C; however, täkō is *reactionary* to the cache (e.g., its callbacks can be triggered on a miss or an eviction) whereas *CachedArrays* is *proactive* and triggers the evictions and insertions before the application accesses the data. Finally, täkō has not been applied to massive 100 GiB DRAM caches and has many of the same downsides as hardware-managed DRAM caches.

III. CachedArrays Details

In this section we discuss the requirements for a high-performing data movement system which does not require deep programmatic changes, the software architecture of our *CachedArrays* design, and end with an example end-to-end design for data movement for training large-scale convolutional neural networks.

A. Requirements for a Data Movement Framework

We observe the following optimizations are important for high-performance data management on heterogeneous memory platforms.

1) Initially allocate data only in one specific device (e.g., fast memory). Hardware caching potentially requires an initial movement from backing memory to the cache, which increases data movement.
2) Elide useless writebacks from one device to another when the data is deallocated. Section V shows this optimization significantly reduces NVRAM writes compared to the hardware managed cache.
3) Move data at a large granularity instead of at the block-level which more efficiently uses memory devices [6], [4]. Section V shows that in CNN applications with a simple policy, *CachedArrays* has higher average memory bus utilization than the hardware managed cache.
4) Avoid polluting the fast memory with data which is rarely referenced, retaining only relevant data in the faster memory.

These optimizations are not possible with fully transparent data management mechanisms such as NUMA and hardware caching. Thus, to generalize access to these optimizations for more applications, we need a *data movement management framework* such that the semantic information from the application can be used to drive the data movement without requiring the application to directly interact with the data movement mechanism.

Only the application (or runtime, compiler, programmer) has the semantic information to know the *future use* of the data which is required for the optimizations above. Specifically, we have found the following semantic information to be the most important to communicate from the application-level to the data manager.

- "I am going to read this object"
- "I am going to write this object"
- "I am not going to access this object for a while"
- "I am never going to access this object again"

B. Data Management API Framework

Our main idea is to separate the three concerns of the data management system: the application's access to data, the data movement mechanism (the data manager), and the data movement policy. By separating these three components, we enable the data movement to use semantic information from the applications without requiring the application to directly interact with the data movement mechanism. Figure 1 presents a high-level overview of a generic heterogeneous memory management system.

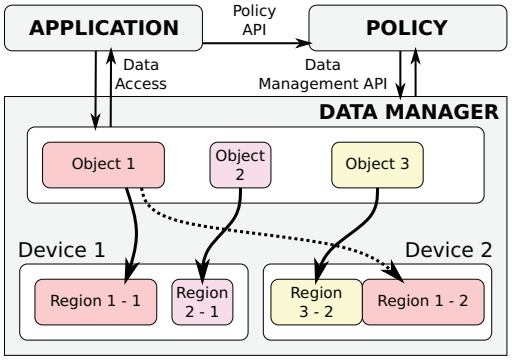


Fig. 1: High level idea of a generic heterogeneous memory management system.

The data access from the application is separated from the other components by a level of indirection (the *object*). The semantic information about the data use is separated from the other components via the *policy API* which the application uses to communicate information about the data use to the policy. Finally, the data movement mechanism is separated from the other components via the *data management API*. An expert programmer or language runtime designer can implement a policy for data movement which directly interacts with the data manager layer to move data without requiring application changes due to object abstraction.

In the rest of this section, we will describe each component of this system in detail.

C. Data Manager

The data manager exposes an API so that the policy can direct the data movement. This *data management API* includes functions to move data, allocate space, query the current state of the system, and update the current state of the system. There are many partial implementations of this API (e.g., täkō [28], *libmemkind*, SMDK). However, they lack fundamental features such as the ability to transparently move data between devices and associate data on multiple devices with a single logical object. Thus, we build *CachedArrays* from the ground up, implementing allocators supporting these operations on top of raw memory obtained either through a single large *malloc* or a memory map from a direct access (DAX) file system.

Objects and Regions: One key observation in prior work to enable the optimizations discussed in Section III-A is that data movement should be done on the "object" level within a program. The object level is where the programmer, compiler, and runtime have specific knowledge of the semantics of the data within their program which can be used to drive data movement and placement considerations. For instance, the programmer knows whether an array will be accessed sparsely or densely, which can affect caching decisions. Examples of "objects" include standard data structures (e.g., standard

containers like `std::vector` in C++) and tensors in CNN-based workloads. In this paper, we focus on CNN-based workloads with relatively large ($>$ 100s of KiB) tensors which are both densely and sparsely accessed.

To support moving objects between different memory pools and construction of higher order constructs like two-level caches we use the idea of *regions*. A region is a contiguous slice of virtual memory that either holds the current data for an object (which we call the *primary region*) or copies of the data for an object (where we call each copy a *secondary*). Two regions are said to be *linked* if they both are associated with the same object. An object's secondary regions may be valid if the primary is *clean* and *read only*, or *stale* if the primary has been updated and without propagating this change to all its secondaries. Our initial prototype of *CachedArrays* requires its underlying memory heaps to be preallocated from the operating system prior to execution (i.e., no dynamic memory allocation from the OS through system calls like *mmap*). Dynamic memory could be added through growing or shrinking the base heap on each device or though multiple heaps per device that are mapped and freed as required. *CachedArrays* inherently supports object reallocation which mitigates fragmentation in either case.

Data Access: In our implementation of *CachedArrays*, we focus on applications which use the kernel programming model (i.e., most computation is done in kernels). The kernels operate on objects by either reading an object, writing an object, and allocating and freeing temporary objects. This model fits well with deep learning frameworks (e.g., Tensorflow [39], PyTorch [40], OneAPI [41], etc.) and many high-performance computing applications.

Because of the kernel programming model, the extra level of indirection for accessing the underlying data of objects can be easily hidden. When the kernel is executed, the runtime can replace the object reference with the current primary region pointer. Unlike OS-based page tables and other "general purpose" indirection tables, the indirection in *CachedArrays* is essentially zero overhead since it happens once for a long-running kernel. One limitation of our current implementation is that an object's primary cannot change during the execution of a kernel. However, we have not observed any examples of prior optimizations which allow for changing the data location while it is actively being accessed.

Data management API: The data manager in *CachedArrays* supports allocation and deallocation of regions, linking and unlinking of regions, high-performance memory copying between regions, and reassignment of an object's primary region.

There are three broad categories of functions: those working on objects, those working on regions, and those working on devices. The former consists of just two functions: **`getprimary`**, to obtain the primary region for an object and **`setprimary`**, to update an object's primary region.

Functions in the second category include **`allocate`** and **`free`** methods for each supported device, as well as a fast memory copy (**`copyto`**) between and within devices. Regions can be linked or unlinked using **`link`** and **`unlink`** respectively. Query functions (**`sizeof`**, **`getlinked`**, and **`in`**) provide methods for obtaining the size of a region, finding a linked region's on a specified device (if one exists), and querying the device to which a region belongs. Regions can be marked and queried as dirty or clean, which helps maintain consistency when an object is associated with multiple regions. Finally, **`parent`** provides a mechanism of going from a region to its object.

D. Data Movement Policy

The goal of the policy is to coordinate the assignment of *objects* to *regions* to improve application performance. The application provides hints to policy regarding how and when *objects* are used. The policy reacts to this information by using the data management API previously described to manipulate the state of objects and regions. This separation of data management from policy is important because (1) it allows the application to influence data placement without needing to understand the low-level details associated with the data management API and (2) allows the policy to be tuned on a per-application basis. In this section, we describe one data movement policy interface which is well suited for very large footprint CNN workloads running on a Cascade Lake heterogeneous memory system with a large NVRAM (Intel Optane DC) and smaller DRAM.

The policy we implement for this class of workloads exposes the functions outlined in Table II. If the application is aware that an object will be used in the near future (for example, when it is about to execute a compute kernel), it can notify the policy with **`will_use`**. More specific hints can be provided with **`will_read`** or **`will_write`** if this is known. This allows the policy to perform preemptive action on the corresponding objects and track object usage. When an object will not be used for some time, it can be marked as **`archived`**. Finally, if the application knows that an object will never be used again, it can use **`retire`**. Only improper use of **`retire`** will affect the correctness of the application. All other functions only influence performance of the application.

How the policy reacts to these hints is an implementation detail of the policy. For example, the policy may respond to all instances of **`will_use`** by prefetching the corresponding object into *fast* memory. In order to drive these decisions, the policy needs to be aware of the relevant characteristics of its memory devices. For a DRAM/NVRAM system, these include:

- Writes to NVRAM are slow and low bandwidth.
- Reads to NVRAM are not much slower than DRAM.
- DRAM bandwidth is high.
- The capacity of NVRAM is large.
- The capacity of DRAM is constrained.

To understand why the policy must be aware of hardware characteristics, we can consider the policy's reaction to **`will_read`** versus **`will_write`**. Because the read bandwidth of NVRAM is relatively high while the write bandwidth is low, the policy may choose to take no action in response to **`will_read`** but move objects into DRAM if needed in response to **`will_write`**. If the properties of the underlying memory change, the policy may need to be modified.

TABLE II: The policy for *CachedArrays* exposes an object-based API for applications to provide hints regarding memory location and movement. These hints can either be placed manually by the programmer or inserted by the compiler.

Operation	Description
`will_use/read/write`	Will read or write in the near future.
`archive`	Will not be used for some time.
`retire`	Will never use again.

```
1  function evict(policy, object)
2    x = DM.getprimary(object)
3    if DM.in(x, FAST)
4      y = DM.getlinked(x, SLOW)
5      sz = DM.sizeof(x)
6      allocated = false
7      if isnothing(y)
8        y = DM.allocate(SLOW, sz)
9        allocated = true
10     end
11     if DM.isdirty(x) || allocated
12       DM.copyto(y, x)
13     end
14     DM.setprimary(object, y)
15     if !allocated
16       DM.unlink(x, y)
17     end
18     DM.free(x)
19   end
20 end
```

Listing 1: An example of building an eviction function from the *CachedArrays* data manager interface (prefixed by `DM`).

To understand how the policy interacts with the data manager, we present the implementation of two kinds of operations the policy may take: evicting and object from *fast* to *slow* memory and prefetching an object from *slow* to *fast* memory. These may be executed in response to the `archived` or the `will_use` hints from the application, respectively. Note that the policy maintains the following invariant: if an object has a region in *fast* memory, then this region will be the primary region for that object.

Listing 1 shows an example implementation of the `evict` operation. If x is already linked with another region in *slow* memory, only a simple copy and free of x_{fast} is needed. Otherwise, x_{slow} needs to be allocated (see lines 4–10). Lines 11–13 show a potential optimization: if we can track whether or not x has been modified while in fast memory then we may be able to elide the expensive copy operation. Line 14 updates the object's primary region y. If a linked region existed in slow memory, x and y need to be unlinked (line 16). This does *not* need to be performed if the slow region was just allocated because the slow and fast regions were then never linked to begin with. Finally, x is freed (line 18).

As a more complicated example, Listing 2 shows an implementation of `prefetch` (e.g., in response to `will_use`). In line 3, the primary region for the object is retrieved. Line 4 checks the device that region resides in: if the region is already in *fast* memory, there is nothing to be done. Line 5 tries to allocate a similar sized region in *fast* memory. If the operation fails (because *fast* memory is full) and the operation is forced, then the policy will forcibly free memory from the *fast* memory. This is done by selecting an initial region

```
1  function prefetch(
2    policy, object, force::Bool = false)
3    x = DM.getprimary(object)
4    if DM.in(x, SLOW)
5      sz = DM.sizeof(object)
6      y = DM.allocate(FAST, sz)
7      if isnothing(y) && force
8        start = find_region(policy)
9        DM.evictfrom(
10         FAST, start, sz
11       ) do region
12         evict(policy, DM.parent(region))
13       end
14       y = DM.allocate(FAST, sz)::Region
15     else
16       return
17     end
18     DM.copyto(y, x)
19     DM.link(x, y)
20     DM.setprimary(object, y)
21   end
22   return
23 end
```

Listing 2: Building a prefetching function out of the data movement API.

`start` via some heuristic like LRU (line 8) and using the function `evictfrom` (lines 9–11) though the data manager to free a contiguous block of memory from the *fast* memory pool of the appropriate size. The `evictfrom` function uses a callback mechanism that allows the policy to use the previously described `evict` to preserve existing objects. Following the eviction, the fast memory allocation is performed again (line 12). Data is copied (line 16), the two regions are linked as siblings (line 17) and the fast memory region is assigned as the primary region (line 18).

We implement and evaluate multiple different policy implementations which take different actions for the same application hints in Section V.

E. End-to-End Example

We use these APIs to implement an end-to-end example of using our split policy-data management interface to optimize data movement for very large-scale CNN training. Our underlying system is a heterogeneous system with DRAM and NVRAM. The important characteristics of the workload that we can take advantage of are the following.

- Each iteration of training is broken into two phases: a forward pass that computes the output of the network and a backward pass that computes the gradients of the model's trainable parameters.
- During the forward pass many intermediate activations are generated that are not consumed until the backward pass.
- These activations are generally used and freed in a first-in last-out manner.

During the forward pass, the convolution reads an input activation tensor, weights, and bias and writes to an output activation tensor. The backward pass kernels require these intermediate activations, which thus must be kept in memory until needed.

For each compute kernel, we call `will_read` on read-only parameters and `will_write` on written parameters, providing

the policy with the opportunity to prefetch data. Following kernel execution on the forward pass, `archive` is called on the weights, bias, and previous activations. A reasonable policy implementation will not eagerly evict data upon an `archive` annotation and will instead prioritize the annotated objects for future eviction if memory pressure is experienced. This means that if the memory required to train the network fits entirely in fast memory, then there is no down side to using `archive`.

Finally, `retire` is handled a little more carefully. For simple linear networks like VGG, intermediate activations can be retired on the backwards pass on a layer-by-layer basis. More complicated networks like ResNet or DenseNet require more precise annotations for which we leverage the Julia language, though we could also leverage APIs in PyTorch or Tensorflow to accomplish this task.

IV. Implementation and Evaluation Methodology

We implemented a software prototype of our data manager in Julia [42], a high-level compiled language targeting technical computing. The main idea behind our implementation is to create an array datatype called a `CachedArray` in Julia backed by objects managed by the data manager. Policy hints take the form of function calls that are forwarded from a `CachedArray` to the manager's policy, which then uses the data management API to manipulate the regions backing the objects. We use Julia's ChainRules [43] package and the Zygote [44] automatic differentiation framework to implement deep learning model training. Semantic annotations are applied when compiling the model with Zygote and therefore don't necessarily require modifications to the original source code. PyTorch 2.0 can provide similar capabilities through the custom compilation of compute graphs.

Next, we describe the general methodology used to evaluate the performance of *CachedArrays*. Unless otherwise specified, our case studies were performed as the sole user of a 2-socket 56 core (112 thread) Intel Xeon Platinum 8276L running Ubuntu 21.10 with 192 GiB (6x32 GiB) DRAM and 1.5 TB (6x256 GiB) Optane DC NVRAM per socket. Experiments were conducted on a single socket, with one thread per core.

To implement deep learning primitives, we wrote a Julia wrapper around Intel's oneDNN [45] library. Memory for kernel input and output tensors comes from the Julia side.

We investigate several different optimizations in the policy to understand their relative impact. These include:

Memory Optimizations (M): We *retire* arrays as soon as possible rather than relying solely on Julia's garbage collector (GC), fulfilling requirement 2 in Section III-A. By doing this, we reduce the amount of data kept alive longer than necessary. If this intermediate data is kept alive, then it must be written to NVRAM if evicted to maintain correctness, resulting in unnecessary slow NVRAM writes. Disabling this optimization means we need to rely on the GC for resource management which involves explicitly triggering collection when memory pressure is detected.

Local Temporary Allocations (L): As discussed in Section III-A (requirement 1), *CachedArrays* has been designed to support unlinked regions in fast memory. In combination with memory optimizations, this is a potent tool for reducing NVRAM traffic. Our policy can be modified to disable local allocations, in which case a newly created array *must* first be allocated in NVRAM and then copied into DRAM before use. The purpose of disabling this optimization is to more directly compare with a naive 2LM implementation. Since 2LM operates as a DRAM cache, each physical cache line must ultimately have a copy in NVRAM. By effectively generating a compulsory miss on first access with *CachedArrays*, we can more closely model the behavior of 2LM.

TABLE III: Large and small CNN models used as benchmarks. Small networks fit entirely in DRAM.

Large Networks			**Small Networks**	
Model	Batchsize	Footprint	Model	Batchsize
DenseNet 264	1536	526 GB	DenseNet 264	504
ResNet 200	2048	529 GB	ResNet 200	640
VGG 416	256	520 GB	VGG 116	320

Prefetching (P): We explore two different policy strategies for responding to `will_read` annotations: one that always prefetches into DRAM and one that never does. The relatively high read bandwidth of NVRAM suggests that kernel execution time may be less sensitive to the location of read-only arguments. Further, prefetching requires making room in DRAM for the prefetched arguments, which could result in other arrays being evicted to NVRAM. By moving data at the block granularity we fulfill requirement 3 in Section III-A.

The combination of optimizations and operating modes investigated is as follows:

- **2LM:** $\emptyset$ – 2LM with no memory optimizations.
- **2LM: M** – 2LM with memory freeing optimizations.
- **CA:** $\emptyset$ – *CachedArrays* with no memory optimizations or prefetching. *All* arrays begin in NVRAM and are moved into DRAM before use, like in a true cache.
- **CA: L** – No memory optimization or data prefetching. Unlike **CA:** $\emptyset$, arrays can be allocated in DRAM only.
- **CA: LM** – Memory optimizations but no data prefetching.
- **CA: LMP** – Memory optimizations *and* prefetching.

A. Large Networks: Comparison With 2LM

The Xeon server used to run these experiments can be configured in two modes, allowing multiple different use cases for NVRAM. The *memory mode* (2LM) configures NVRAM as main memory with DRAM serving as a transparent, direct-mapped hardware managed cache [4]. *App direct* allows NVRAM to be used directly. In this mode, NVRAM can either be configured as an extra NUMA node to be used automatically by the OS, or mounted as a direct access (DAX) file system. In the latter case, files memory mapped from the NVRAM-based DAX file system are directly mapped into the program's address space with reads and writes sent directly to the NVRAM devices. Our *CachedArrays* based policies uses this last option.

To compare against 2LM, the overall memory footprint of the models used for benchmarking must greatly exceed the

size of DRAM cache on a single socket which we accomplish though a combination of deep networks and large batchsize. For large traditional networks, we used ResNet 200 [46] and DenseNet 264 [47], two networks with complex data flow. As an extra comparison point, we implemented VGG 416 [9], which is essentially a greatly extended VGG 16. A summary is provided in Table III along with the approximate minimum memory footprint required for a single iteration of training. The batchsizes for the small networks were chosen such that the memory requirement for training was between 170 GB and 180 GB: small enough to fit entirely in DRAM.

All runs in 2LM were conducted with a maximum memory size of 1300 GB. When we run in *app direct* mode, *CachedArrays* is configured with the same hardware limits of 180 GB DRAM and 1300 GB NVRAM. After each training iteration (forward + backward pass), the GC was invoked to clean up all temporary memory, leaving only the model weights and computed gradients. The local heap was then defragmented before the next run to help keep behavior similar across iterations (defragmentation overhead is negligible compared to the iteration time). Each model was run for four iterations and performance metrics were checked to ensure that behavior for each iteration was consistent. Input data was randomly generated using a normal distribution.

For each experiment, we used hardware performance counters to capture read and write traffic to DRAM and NVRAM. For the 2LM based runs, the same hardware counters were used to also capture DRAM cache statistics including cache hits, clean cache misses, and dirty cache misses. All memory heaps used by *CachedArrays* are pre-allocated before running our experiments, ensuring that the OS assigns physical pages to all virtual pages within the heap. This in itself provides a large speedup over Julia's default allocator. For large allocations, normal allocators typically memory map new memory from the OS, which must be zero-initialized. Due to this overhead, we use 2LM with the *CachedArrays* allocator as the baseline.

B. Small Networks - Sensitivity Analysis

Since we control data movement from software, we can vary the amount of DRAM available to *CachedArrays* to see how our simple policy holds up as we decrease the DRAM allowance. To that end, we use the small networks and batch sizes provided in Table III. These are chosen such that the memory footprint required for training is between 170 and 180 GB and thus can fit within the DRAM of a single socket of our benchmark machine. We then vary the DRAM budget from the full 180 GB down to 0 GB (NVRAM only). As for the large benchmarks, we run each model for four iterations, defragmenting heaps between iterations and checking that variance between iterations was minimal. For this experiment, we use VGG 116 instead of VGG 416 in order to maintain a higher batch size. All of these runs were conducted in the **CA: LM** mode as this performs well across all networks.

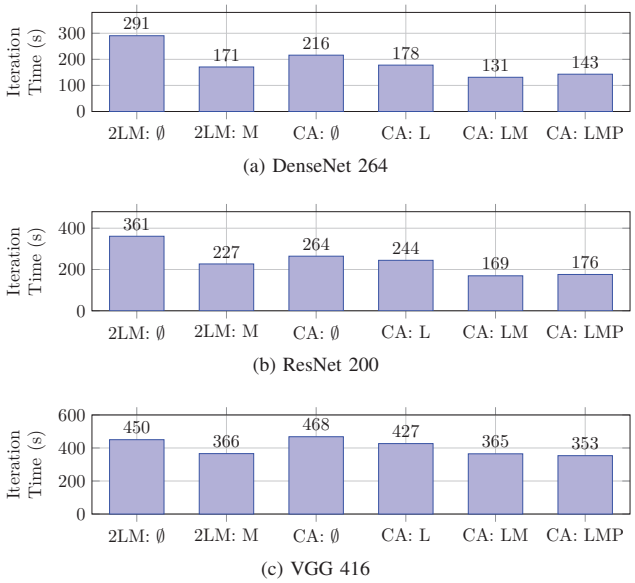


Fig. 2: Average execution time for a single iteration of training for the large networks, categorized by operating mode and applied optimizations.

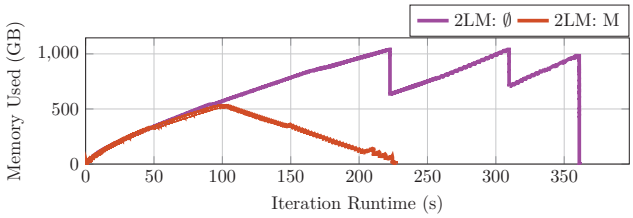


Fig. 3: Resident heap memory through a single iteration of ResNet training. The 2LM based experiments have a memory heap which is implicitly managed by Intel's DRAM cache.

V. Results

Figure 2 shows the absolute runtime for a single iteration for the large CNN benchmarks. First, we explore the performance of **2LM: ∅** (hardware DRAM cache with no optimizations) and **2LM: M** (DRAM cache with memory optimizations). For *CachedArrays*, **CA: L** (supporting local-only allocation) is faster than **CA: ∅**, and applying memory optimizations further improves performance. Prefetching hurts performance for DenseNet and ResNet, but improves performance for VGG.

a) Why does semantic information improve performance of the hardware DRAM cache?: Figure 2 shows that adding the memory freeing optimizations improve the performance for 2LM as well as *CachedArrays*. To understand why, we investigate the physical addresses and cache behavior for ResNet 200. Figure 3 shows the occupancy of the memory heaps for a single iteration of ResNet under the two 2LM operating regimes. These runs only have a single memory heap that is implicitly managed by the hardware DRAM cache. Without any memory optimizations (**2LM: ∅**), memory usage

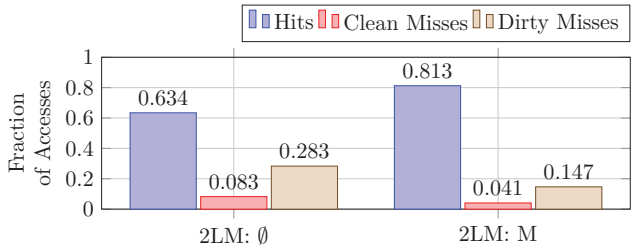
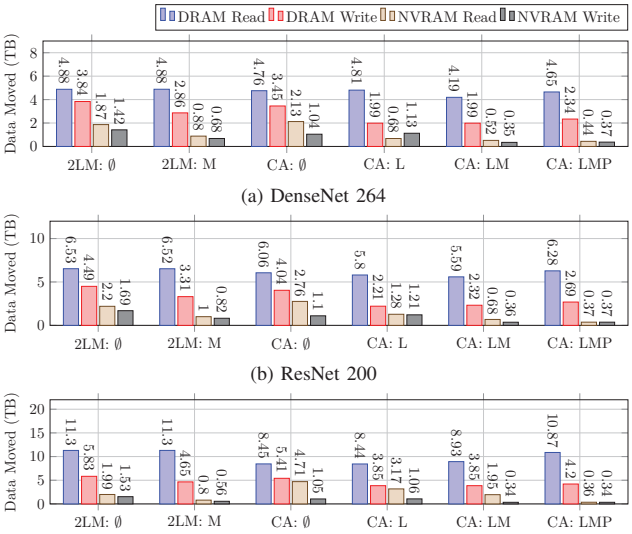


Fig. 4: DRAM cache tag statistics for a single training iteration of ResNet 200.



(a) DenseNet 264

(b) ResNet 200

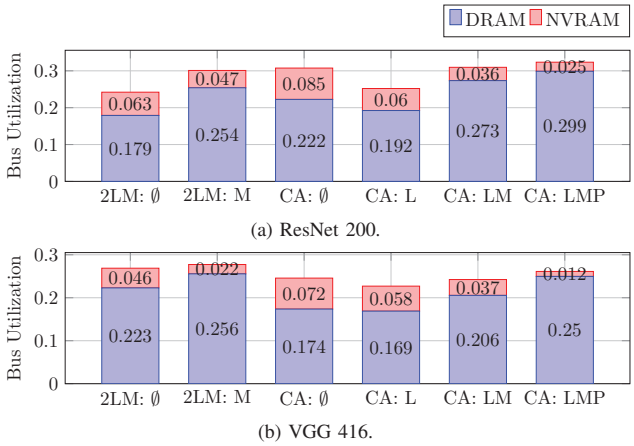
(c) VGG 416

Fig. 5: Amount of data moved for a single iteration training for the large CNNs.

keeps increasing until the garbage collector is run (around time 220s) which causes the monotonically increasing behavior of that curve. Note that both PyTorch and Tensorflow must also run their own garbage collectors to free GPU memory. In contrast, when the annotated run (**2LM: M**) begins its backward pass, (around time 100s) it proactively frees memory produced on the forward pass. This results in more reuse of the physical pages backing that memory, leading to fewer cache misses and less data movement.

Supporting this idea is Figure 4, which shows the average DRAM cache hit, clean miss, and dirty miss rates for the two 2LM runs. The annotated run has an 18% higher hit rate and 50% lower dirty miss rate, both of which improve 2LM performance [4]. By adding semantic information, we achieve better utilization of the data movement mechanism.

b) Why does CachedArrays *outperform 2LM?:* To understand the performance difference between 2LM and *CachedArrays*, we first focus on the total amount of memory moved for a single iteration of training. Figure 5 shows the total amount of DRAM and NVRAM traffic for a single iteration of training for two of our large networks, further broken into reads and



(a) ResNet 200.

(b) VGG 416.

Fig. 6: Average utilization of the DRAM bus.

writes. With no optimizations, *CachedArrays* is slower than memory-optimized 2LM and in the case of VGG is even slower than unoptimized 2LM. For DenseNet and ResNet, **CA: ∅** generates similar read and write traffic to **2LM: ∅**, though with generally fewer NVRAM writes. The saving of NVRAM writes occurs because even though memory optimizations are not applied, we still run the garbage collector after every iteration of training. In 2LM, this optimization does not help because physical addresses used on the backwards pass are dirty with respect to the DRAM cache.

Even though this still applies to VGG, **CA: ∅** is still slower than **2LM: ∅**. This is where *traffic shaping* comes into play. Prior work shows that large sequential accesses provide the highest bandwidth for NVRAM [6], [4]. In 2LM, NVRAM traffic is haphazard and results from conflict misses in the DRAM cache. With *CachedArrays*, NVRAM traffic is the result of explicit, well-shaped memory copies.

Figure 6 shows the average utilization of the memory bus for a single iteration of training for ResNet 200 and VGG 416. For ResNet, **CA: ∅** achieves a higher average utilization than **2LM: ∅** while the situation is reversed for VGG. The memory movement engine in *CachedArrays* is highly multi-threaded, specifically targeting large memory sizes which works well for ResNet because the large batch size of 2048 results in large memory transfers. However, a much smaller batch size of 256 is used for VGG, leading to smaller data transfers and more parallelization overhead. However, bus utilization must be considered in the context of overall memory moved. As optimizations are applied via *CachedArrays*, bus utilization tends to increase and overall traffic generated tends to decrease, both resulting in better performance.

Impact of Local Allocation: As shown in Figure 5, adding the local allocation optimization to *CachedArrays* significantly decreases the NVRAM read and DRAM write traffic due to the elision of the initial memory copy. The performance difference between these two is largely due to the decreased time spent synchronously moving data.

Impact of Memory Optimizations: Memory optimizations

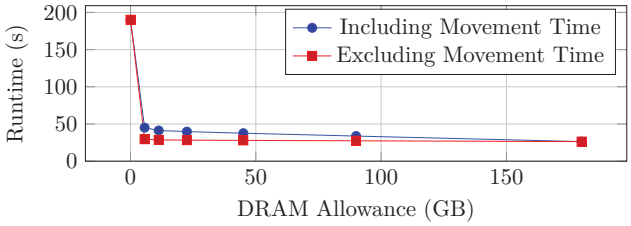


Fig. 7: Average runtime for a single training iteration for the small DenseNet. Absolute wall clock time (blue) and time without data movement (red). Runs with local allocation and memory optimizations, but *without* prefetching enabled.

decrease memory pressure by freeing memory as soon as possible. This avoids many unnecessary writes to NVRAM, which can be seen in Figure 5. In particular, observe the difference in NVRAM reads and writes in Figure 5a between **CA: L** and **CA: LM** for DenseNet. For **CA: L**, the number of NVRAM writes exceeds the number of NVRAM reads, implying that *unnecessary* data is being moved to NVRAM. This is the result of intermediate allocations not being freed as soon as possible. When applying memory optimizations (**CA: LM**), the number of NVRAM writes for DenseNet drops from about 1100 GB to about 350 GB, with NVRAM reads exceeding NVRAM writes. The other networks experience similar decreases in NVRAM writes when applying memory optimizations. The local allocation and memory optimizations reduce the amount of NVRAM writes down to a bare-minimum.

Impact of Prefetching: Enabling data prefetching harms performance for DenseNet and ResNet. As can be seen in Figure 5, prefetching decreases NVRAM read traffic and increases DRAM read traffic because arrays are moved from NVRAM to DRAM where there are referenced multiple times to compute the backwards pass. However, some operations are not sensitive to the bandwidth of their read-only arguments. Hence, this prefetch wastes time and potentially evicts other arrays. In the case of VGG, on the other hand, prefetching *does* slightly improve performance since it significantly decreases NVRAM read traffic by a factor of 5.4×. Thus, there is no "one size fits all" approach to memory management.

In summary, full *CachedArrays* results in less DRAM and NVRAM traffic overall, because it is aware that data freed on the backwards pass is semantically dead, and thus does not need to be written back to NVRAM. Hardware caches do not have this semantic insight and thus must always act conservatively. Furthermore, Figure 6 shows the average DRAM bus utilization though out an iteration of training. *CachedArrays* maintains a higher average utilization while also moving less total data. *CachedArrays* both maintains the memory semantics required to elide unnecessary dirty writebacks and uses traffic shaping to achieve high bandwidth.

c) Sensitivity to DRAM capacity: The runtime for a single iteration of training for the small DenseNet is shown in Figure 7. Running with only NVRAM results in a 3–4× performance penalty (other models show similar results). However, with even a small amount DRAM, *CachedArrays* is able to get most of that performance back.

Figure 7 also shows what the performance would be if *CachedArrays* had perfectly asynchronous data movement (as opposed to purely synchronous) and could overlap movement with execution. Asynchronous data movement could be implemented with a separate thread pool or with an accelerator [48]. For DenseNet and ResNet, this projected performance varies only slightly as the DRAM budget decreases. VGG, on the other hand, still experiences a slow-down due to more reads being generated to NVRAM. These results are consistent with the large network results where DenseNet and ResNet had lower performance with prefetching unlike VGG. The kernels composing VGG are more sensitive to read bandwidth.

d) Why is a small amount of DRAM enough?: Contrary to the behavior of pure DRAM, DRAM to NVRAM copy bandwidth actually *decreases* with increasing parallelism [6], [4]. Furthermore, the copy kernel in *CachedArrays* uses *non-temporal* stores to NVRAM, which are crucial for best performance. OneDNN kernels are *not* optimized for writing to NVRAM and thus the performance with all NVRAM is slow. When a small amount of DRAM is used, the output parameters of the computation kernels can be placed in DRAM and any movement of data from DRAM to NVRAM goes through code-paths optimized to get the best write performance out of NVRAM. An interesting area for future research would be to explore computation kernel implementations that specialize based on the memory location of its arguments (much like existing specialization via just-in-time compilation based on the dimensions of the arguments).

VI. *CachedArrays* Extensions

In this section, we will discuss three directions for extensions of *CachedArrays*: (1) supporting sparse data structures and more complex deep learning workloads, (2) supporting other heterogeneous memory devices, and (3) supporting application frameworks beyond Julia. The approach taken by *CachedArrays* is not limited to static computation graphs or the well understood data use patterns of CNN workloads like many previous works [9], [8]. The *CachedArrays* policy responds to runtime annotations, and can apply to applications exhibiting dynamic memory use such as Transformers, RNNs, and Mixtures of Experts [49]. For example, Hildebrand et al. applied a dynamic policy to a DLRM workload [50] and found that flexibility in the data movement policy is required to meet different memory access patterns as the locality of the data changes based on user input [15]. Further, we could augment the policy with real-time kernel performance information, allowing the policy to explore and adapt its strategy.

Additionally, our framework is agnostic to the compute/interconnect framework surrounding the memory. Thus, *CachedArrays* applicable to other heterogeneous memory devices such as local/remote memory (e.g., CXL) and CPU/GPU memory. In this paper, we focused on a realistic platform available for experimentation, but the *CachedArrays* framework is not limited to DRAM+NVRAM systems. In fact, the flexibility enabled by separating the data movement policy from the

data movement mechanism means that when migrating an application to a new heterogeneous memory platform, the user-defined policy does not have to be modified. The only change necessary is for the platform developer to provide the interface needed to implement the policy.

Finally, while we implemented an initial prototype of *CachedArrays* in Julia, the approach is not limited to Julia. Frameworks such as PyTorch [40] and Tensorflow [51] have very similar interfaces that can be exploited to implement *CachedArrays*. For instance, PyTorch has a `torch.Tensor` class that is very similar to Julia's `Array` type and it supports on-the-fly compiler transformations with `torch.compile`.

VII. Conclusions

In this paper, we presented *CachedArrays*, a framework for managing data movement in heterogeneous memory systems. *CachedArrays* is a software framework that allows the user to define a data movement policy that is applied at runtime. We implemented *CachedArrays* in Julia and evaluated it on a DRAM+NVRAM system and found significant performance improvements compared to the hardware-implemented cache. Futhermore, although not evaluated in this paper, *CachedArrays* can be extended to other heterogeneous memory systems, frameworks beyond Julia, and applications beyond CNNs.

References

[1] S. Van Doren, "Abstract - HOTI 2019: Compute express link," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2019, pp. 18–18.

[2] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[3] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 598–611.

[4] M. Hildebrand, J. T. Angeles, J. Lowe-Power, and V. Akella, "A case against hardware managed DRAM caches for NVRAM based systems," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 194–204.

[5] J. T. Angeles, M. Hildebrand, V. Akella, and J. Lowe-Power, "Investigating hardware caches for terabyte-scale NVDIMMs," in *Non-volatile Memories Workshop (NVMW 2021)*, 2021.

[6] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[7] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun, "Sage: Parallel semi-asymmetric graph algorithms for NVRAMs," *Proc. VLDB Endow.*, vol. 13, no. 9, p. 1598–1613, May 2020. [Online]. Available: https://doi.org/10.14778/3397230.3397251

[8] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 875–890. [Online]. Available: https://doi.org/10.1145/3373376.3378465

[9] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 18:1–18:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195660

[10] X. Chen, D. Z. Chen, and X. S. Hu, "moDNN: Memory optimal DNN training on GPUs," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 13–18.

[11] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[12] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: https://doi.org/10.1145/3394486.3406703

[13] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale model training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 551–564. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/ren-jie

[14] S.-F. Lin, Y.-J. Chen, H.-Y. Cheng, and C.-L. Yang, "Tensor movement orchestration in multi-GPU training systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1140–1152.

[15] M. Hildebrand, J. Lowe-Power, and V. Akella, "Efficient large scale DLRM implementation on heterogeneous memory systems," in *High Performance Computing: 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21–25, 2023, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 42–61. [Online]. Available: https://doi.org/10.1007/978-3-031-32041-5_3

[16] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, 2019, pp. 331–345. [Online]. Available: https://doi.org/10.1145/3297858.3304024

[17] S. Kannan, Y. Ren, and A. Bhattacharjee, "KLOCs: kernel-level object contexts for heterogeneous memory systems," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 65–78.

[18] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, 2017, pp. 631–644. [Online]. Available: https://doi.org/10.1145/3037697.3037706

[19] S. Bergman, P. Faldu, B. Grot, L. Vilanova, and M. Silberstein, "Reconsidering os memory optimizations in the presence of disaggregated memory," in *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, 2022, pp. 1–14.

[20] B. Peng, Y. Dong, J. Yao, F. Wu, and H. Guan, "Flexhm: A practical system for heterogeneous memory with flexible and efficient performance optimizations," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, pp. 1–26, 2022.

[21] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for multi-tiered memory systems," in *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, 2021, pp. 715–728.

[22] T. D. Doudali, D. Zahka, and A. Gavrilovska, "Cori: Dancing to the right beat of periodic data movements over hybrid memory systems," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 350–359.

[23] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, 2021, pp. 392–407.

[24] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent page placement for CXL-enabled tiered-memory," in

Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2023, pp. 742–755.

[25] H. Li, K. Liu, T. Liang, Z. Li, T. Lu, H. Yuan, Y. Xia, Y. Bao, M. Chen, and Y. Shan, "HoPP: Hardware-software co-designed page prefetching for disaggregated memory," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1168–1181.

[26] S. Kumar, A. Prasad, S. R. Sarangi, and S. Subramoney, "Radiant: efficient page table management for tiered memory systems," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, 2021, pp. 66–79.

[27] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 79–92.

[28] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täkō: a polymorphic cache hierarchy for general-purpose optimization of data movement." in *ISCA*, 2022, pp. 42–58.

[29] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems, MLSys*, vol. 3, pp. 711–732, 2021.

[30] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[31] O. Rausch, T. Ben-Nun, N. Dryden, A. Ivanov, S. Li, and T. Hoefler, "A data-centric optimization framework for machine learning," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.

[32] C. Giannoula, K. Huang, J. Tang, N. Koziris, G. Goumas, Z. Chishti, and N. Vijaykumar, "DaeMon: Architectural support for efficient data movement in fully disaggregated systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 1, pp. 1–36, 2023.

[33] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri, "Write-avoiding algorithms," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 648–658.

[34] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1166–1177, 2017.

[35] W. Pan, T. Xie, and X. Song, "Hart: A concurrent hash-assisted radix tree for DRAM-PM hybrid memory systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 921–931.

[36] Y. Shen and Z. Zou, "Efficient subgraph matching on non-volatile memory," in *International Conference on Web Information Systems Engineering*. Springer, 2017, pp. 457–471.

[37] J. Jung, J. Kim, and J. Lee, "DeepUM: Tensor migration and prefetching in unified memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 207–221.

[38] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: OS design for heterogeneous memory management in datacenter," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 521–534.

[39] Tensorflow, https://www.tensorflow.org.

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[41] oneAPI, https://www.oneapi.io.

[42] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

[43] F. Schäfer, M. Tarek, L. White, and C. Rackauckas, "AbstractDifferentiation.jl: Backend-agnostic differentiable programming in Julia," 2021.

[44] M. Innes, "Don't unroll adjoint: Differentiating SSA-form programs," *CoRR*, vol. abs/1810.07951, 2018. [Online]. Available: http://arxiv.org/abs/1810.07951

[45] oneDNN, https://github.com/oneapi-src/oneDNN.

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[47] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[48] A. Biswas, "Sapphire rapids," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–22.

[49] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *CoRR*, vol. abs/1701.06538, 2017. [Online]. Available: http://arxiv.org/abs/1701.06538

[50] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: http://arxiv.org/abs/1906.00091

[51] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi