Application-Attuned Memory Management for Containerized HPC Workflows

Moiz Arif[†], Avinash Maurya[†], M. Mustafa Rafique[†], Dimitrios S. Nikolopoulos[λ], Ali R. Butt[λ]
[†]Rochester Institute of Technology, [λ]Virginia Tech
[†]{ma3890, am6429, mrafique}@cs.rit.edu, [λ]{dsn, butta}@cs.vt.edu

***Abstract*—High-Performance Computing (HPC) jobs consist of data and memory-intensive tasks often executed as workflows or ensembles to facilitate efficient and coordinated execution. These workflows are traditionally executed on HPC systems and have unique memory requirements based on the data size, computational complexity, and I/O activity. Recently containerized execution of these workflows has been extensively explored. Containerized workflow execution of HPC jobs requires several terabytes of memory that exceed node capacity, resulting in excessive data swapping to slower storage, degraded job performance, and failures. Similarly, colocated bandwidth-intensive, latency-sensitive, or short-lived workflows suffer from degraded performance due to contention, memory exhaustion, and higher access latency due to suboptimal memory allocation. Recently, tiered memory systems comprising persistent memory and compute express link (CXL) have been explored to provide additional memory capacity and bandwidth to memory-constrained systems and applications. However, current memory allocation and management techniques for tiered memory subsystems are inadequate to meet the diverse needs of colocated containerized jobs in HPC systems that concurrently run workflows and ensembles at scale. This paper leverages tiered memory systems for containerized HPC workflows and proposes efficient memory management policies including intelligent page placement and eviction policies to improve memory access performance. Our page allocation and replacement policies incorporate task characteristics and enable efficient memory sharing between workflows. We integrate our policies with the popular HPC scheduler, SLURM, and container runtime, Singularity, to show that our approach improves tiered memory utilization and application performance and reduces workflow execution times by up to 51%, 87%, and 35% as compared to the ideal, realistic, and optimized tiered execution environments, respectively.**

***Index Terms*—Tiered Memory Systems, Memory Management, Containers, HPC Workflows**

I. Introduction

High-Performance Computing (HPC) jobs are composed of compute and memory-intensive tasks that involve large-scale simulations, data analysis, or scientific computations. These jobs can be managed and executed as HPC workflows, composed of a series of interconnected computational tasks, often represented as directed acyclic graphs (DAGs), executed in a specific order to achieve a larger computational goal, where the output of one task serves as the input for subsequent tasks. HPC workflows are complex and involve diverse computational tasks, including data pre-processing, compression, checkpointing, simulations, and post-processing. Similarly, workflows may also consist of HPC ensembles, which involve the simultaneous execution of multiple instances of a task where each ensemble member represents a different realization of the task using different input parameters, initial conditions, or algorithmic choices. HPC ensembles are often used in scientific simulations [1], [2], climate modeling [3] computational physics [4], optimization problems [5], and machine learning [6] to explore a broader range of possibilities and derive more robust conclusions. Moreover, HPC workloads are deconstructed into smaller workflows, which enable node-level colocation on HPC systems, optimize resource utilization, and address stranded memory problems. By effectively managing and executing HPC jobs using workflows and ensembles, researchers can achieve improved scalability, optimize resource utilization, analyze uncertainties, and gain valuable insights from their computational endeavors.

Traditionally, HPC workflows and ensembles are executed on bare-metal HPC systems to leverage high-performance on specialized hardware. However, in recent years, using containers [7]–[9] for executing HPC workflows is being explored for improved portability and reproducibility. Containerization technologies, such as Singularity [10], enables efficient resource utilization by colocating multiple workflows on the same host, thus taking advantage of increased parallelism and throughput. Because of this, leading computing facilities, such as Riken Fugaku [11], OLCF Summit [12], ALCF Theta [13] and ALCF Polaris [14], have adopted Singularity as the containerization runtime. While containerized environments lead to higher system utilization and energy efficiency due to colocation, they also incur performance penalties [15]–[18] due to diverse memory requirements, latency sensitivity, throughput, and resource sharing. Similarly, colocated workflows have complex performance requirements related to memory such as access latency, bandwidth and capacity. Moreover, short-lived workflows require minimum access latency and high bandwidth to meet SLAs. Traditional memory management techniques do not account for these requirements leading to suboptimal memory allocation amongst colocated workflows. These challenges necessitate the development of holistic application-attuned memory management techniques to efficiently utilize containers for running HPC workflows for increasing overall memory utilization, improving workflow performance, and reducing the startup time of large-scale containerized HPC deployments.

Memory tiers in modern HPC systems [19]–[21], such as those incorporating DRAM and persistent memory play a crucial role where each tier offers different performance, ca-

pacity, bandwidth, and access latency. Similarly, CXL memory introduces a new tier into the existing memory hierarchy providing direct memory access (DMA) via load/store semantics for efficient data movement across the memory hierarchy. Managing memory in a tiered system involves employing various strategies to optimize data placement and movement across these tiers [22]–[24] such as interleaving [25], weighted interleaving [26], and AutoNUMA [27] thereby enhancing overall system performance. Additionally, data movement policies are employed to dynamically move data between tiers based on access patterns, while ensuring that data is stored in the most appropriate tier at any given time. These techniques work well for general-purpose computing workloads, however, they cannot optimize memory placement and management across tiers based on the workflow requirements. Similarly, the colocation of diverse workflows from various HPC workloads renders approaches, such as AutoNUMA, TPP, weighted interleaving, etc. sub-optimal on a tiered memory system since these approaches do not differentiate between workload pages and their sensitivity to page movement between tiers. Thus, the performance of workflows is negatively impacted by inefficient allocation and management of tiered memory resulting in: (1) limited access bandwidth; (2) suboptimal memory capacity utilization; and (3) higher memory access latencies.

In this paper, we address the aforementioned limitations of containerized HPC workflows and address the performance challenges in tiered memory systems by proposing application-attuned intelligent memory management policies. Our memory management policies incorporate the access latency associated with memory tiers to optimize the performance of workflows while incorporating the performance characteristics, i.e., sensitivity to latency, bandwidth, and capacity, of each workflow task. Our policies leverage workflow memory access patterns and system memory utilization to evict data from memory tiers. Our proposed memory management policies support containerized and bare-metal executions, however, the colocation of containerized HPC workflows introduces significant complexity due to diverse memory resource requirements of each colocated workflow compared to the bare-metal HPC execution model. To demonstrate the effectiveness of our approach, we integrate it with SLURM and Singularity [10], and conduct extensive evaluations using real-world HPC jobs. To the best of our knowledge, this is the first effort to address the performance challenges of leveraging tiered memory to provide additional memory to containerized HPC jobs.

Specifically, we make the following contributions:

- We analyze the memory requirements of popular HPC workflows and ensembles and explore challenges related to the suboptimal memory management and utilization on the performance of HPC workflow and ensembles.
- We propose holistic application-attuned memory management policies for tiered memory for containers that optimizes access latency, bandwidth, and capacity to enable efficient workflow execution with minimal overhead.
- We implement and integrate our approach and runtime with HPC job scheduler, i.e., SLURM, and container runtime, i.e., Singularity, to support several classes of workflows that allocate additional memory from various memory tiers including local DRAM, CXL (emulated through remote NUMA), and persistent memory (PMem) modules.
- We thoroughly evaluate the proposed memory management policies and show that our approach increases overall memory utilization, improves workflow performance and reduces the startup time for large-scale containerized HPC deployments. Our approach shows up to 35% performance improvement over the existing workflow oblivious tiered memory management techniques.

II. Background and Motivation

A. HPC Workflows and Workflow Management Systems

HPC workloads are composed of a series of tasks, organized as workflows, that work in tandem to run larger scientific applications such as (1) scientific simulations, which run in embarrassingly parallel or tightly coupled fashion; (2) surrogate computations, which typically generate a deep-learning-based approximation to assist the scientific simulation for faster convergence; (3) real-time data analysis, which includes on-the-fly data manipulation and visualization based on which experiments and/or algorithms are steered; (4) producer-consumer workflow patterns, where workflows consume data generated by other workflows; and (5) checkpointing for fault-tolerance, posthoc analysis, supporting out-of-core adjoint computations, or explaining the evolution of data and scientific model. Several Workflow Management Systems (WMS), e.g., Pegasus [28], Cromwell [29], and Nextflow [30], facilitate the orchestration and automation of complex computational workflows. WMSs interact with sophisticated schedulers to efficiently allocate computing resources, optimize task dependencies, and balance workflows. However, they face challenges in managing diverse workflows with varying resource demands, adapting to dynamic system conditions, and ensuring optimal resource utilization amidst changing priorities and constraints [31]. Additionally, optimizing memory allocation in tiered memory systems, efficient data movement between memory tiers, optimal data placement, catering for data locality, memory requirements, and inter-task communication further complicates the scheduling process and is not supported in modern WMSs [28], [29], [32].

B. Memory Characteristics for HPC Jobs

HPC jobs pose diverse requirements to memory subsystems, such as combinations of large memory tiers, low latency, and high bandwidth. These requirements can change dynamically during job execution. Moreover, HPC jobs are often composed of several workflows with diverse memory requirements [33]–[35] causing memory starvation, contention, and degraded performance. Traditionally, the basic allocation unit for HPC jobs is a compute node that leads to reduced resource utilization and fragmentation. The available memory is limited by the job-level allocations and the total physical memory installed

on each server. To improve the performance of HPC jobs, in-memory computation is becoming increasingly popular [36] leading to higher memory demands in HPC clusters.

In containerized execution, memory is allocated at the start based on the memory requirement of the job and does not support dynamic memory allocation based on different execution phases of HPC workflows. Typically, HPC jobs are deployed as separate workflows [37] with diverse resource profiles, e.g., compute and memory-intensive tasks, bandwidth-intensive operations, and capacity- and latency-sensitive operations. Given the varying demands of different resource profiles, accurately identifying the memory requirements for each workflow is challenging. Similarly, colocated containerized HPC workflows and ensembles have additional resource limitations, e.g., CPU, memory, storage, I/O, and network, which are specified by the workflow and negatively impact its performance. These restrictions limit the performance of highly parallel memory and data-intensive workflows where most tasks require a large amount of memory to store the input, intermediate, and output data of various tasks of HPC workflow. Similarly, accurately estimating the memory requirements of workflow tasks to allocate enough memory is challenging resulting in a loss of critical computation during failures [38].

C. Tiered Memory Systems

Tiered memory systems utilize the latest advancements in memory subsystems to provide large memory to servers and workflows. It allows workflows to scale by utilizing additional memory available beyond the total available DRAM on each server. In tiered memory systems, the DRAM tier is utilized for high-speed, low-latency access to frequently accessed data, whereas PMem [39] bridges the gap between volatile and non-volatile memory to provide a balance between speed and persistence. Recently CXL [40], [41] has been explored to provide high-speed, low-latency memory access between the host processor and devices while expanding overall memory capacity and bandwidth [25], [42]. CXL memory also enables direct access to additional memory resources and optimizes data movement by providing byte-addressable, cache-coherent memory in the same physical address space and allowing transparent memory allocation using standard memory allocation APIs. Even with colocated memory-intensive tasks, HPC jobs rarely use the entire allocated memory and often leave a large amount of unused memory during their life cycles. For instance, our evaluations (Section IV) demonstrate that during the initial 120 seconds of training BERT [43] model, ∼55%-80% of the allocated memory remains idle, thereby becoming *cold memory* pages. Moving these cold memory pages to a slower memory tier can allow hot memory pages to reside in fast memory tiers and improve application performance. Intelligently placing data between different memory tiers based on their access pattern ensures that frequently used data remains in high-speed low-latency memory tiers, thereby minimizing the need for costly swaps to slower persistent storage.

Figure 1 shows the impact of allocating tiered memory to different containerized workflows. The setup consists of

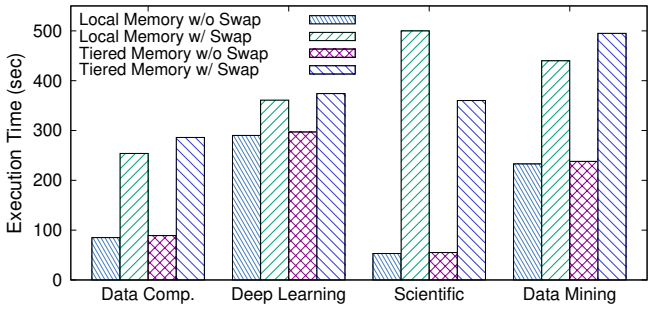

Fig. 1: Impact of tiered memory on workflows with SSD-based swap.

512 GB of main memory, 1 TB persistent memory, and emulated CXL memory using a remote NUMA socket (detailed in Section IV-C). The performance of all workflows significantly drops when onboard system memory is limited and memory pages are swapped to disk-based swap storage. Allocating memory from different tiers improves the performance of each workflow regardless of the workload type and memory access pattern, however, bandwidth-intensive tasks benefit more due to additional bandwidth available over the CXL interface. Moreover, the performance is further improved when the memory pages are actively migrated to CXL-based memory instead of disk-based swap storage.

With the popularity of containerized HPC workflows, there is a need to rethink the management of tiered memory to support granular memory allocation for workflow tasks, intelligent data placement techniques for latency-sensitive tasks, and enable fast data sharing between local and remote tasks from the same or different workflows to increase the resource utilization and reduce the execution time of HPC jobs. To the best of our knowledge, we are the first to explore tiered memory for containerized HPC jobs and propose specialized memory allocation and management policies to meet latency, bandwidth, and capacity requirements of workflow tasks.

III. Tiered Memory Management for HPC Jobs

In this section, we outline the design objectives and detail our proposed memory management policies for optimizing workflow execution on HPC systems utilizing tiered memory.

A. Design Objectives

The main goal of our proposed memory management policies and runtime is to minimize the execution time of HPC workflows by mitigating the impact of inefficient memory allocations, replacement, and movement policies of existing tiered memory approaches. The key objectives of the proposed policies are as follows:

1) Design intelligent memory management policies to fully utilize distributed heterogeneous memory subsystems to improve the overall memory utilization, and reduce workflow failures due to limited memory [44], [45], thus improving the overall system throughput.
2) Mitigate the impacts of using tiered memory on workflow performance by using intelligent page allocation

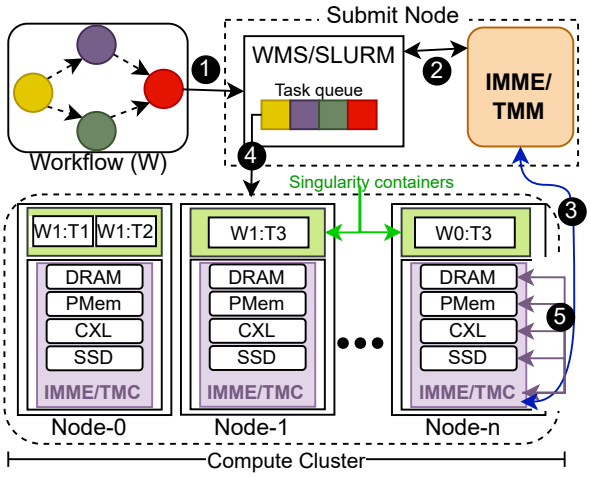

Fig. 2: High-level system architecture with IMME leveraging tiered memory for containerized HPC workflows.

and replacement policies that leverage the access latencies of different memory tiers, the interconnection bandwidth, and local memory availability.

3) Design and develop a highly efficient and lightweight runtime that manages allocation and movement of additional memory requests from HPC workflows and transparently moves memory pages between memory tiers to maximize the overall system performance.
4) Enable workflows to use tiered memory with minimal overhead and modifications to the user code.

B. High-level System Overview

The high-level architecture of our proposed runtime is shown in Figure 2. The workflow is first submitted to the WMS where it is converted to an executable workflow represented by a DAG. Our proposed runtime ensures that the HPC workflows optimally leverage additional memory from the memory tiers and enable workflow-aware memory allocation to jobs. Workflow containers can request memory from specific memory tiers which can be different from the initial memory allocation. Our allocation policy serves such memory requests by efficiently allocating memory pages from the requested memory tier. It identifies the best memory tier based on the workflow characteristics, i.e., latency sensitivity, bandwidth and capacity intensive, and execution makespan, and allocates either the entire block from a single tier or from multiple memory tiers including the local and CXL memory. Our target capacity-intensive jobs, such as training DL models [46], [47] and large-scale simulations [48], require large memory capacity for continued execution and are independent of their latency and bandwidth requirements. If enough local memory is not available, then our page replacement policy and proactive swapping mechanism move existing memory pages to the appropriate lower memory tiers to provide large contiguous memory space for workflows.

1) System Characteristics: In this paper, we use heterogeneous memory systems that include at least two memory tiers including the DRAM, PMem, and CXL memory supported by NVMe, SSD-based storage, and similar technologies in memory and storage subsystems. We note that distributed HPC execution environments, such as Aurora [19], and Pegasus [21], host a subset of these memory and storage tiers at each server. We also assume that the tiered memory is accessible on every node in the cluster including PMem and CXL memory over the CXL interconnect, which provides high-bandwidth, low-latency, cache-coherent access to memory resources and supports multiple memory types, including DDR5 [49], and HBM [40]. Similarly, multiple tasks of a workflow can be scheduled on a server to achieve higher system throughput and improve workflow performance.

TABLE I: APIs to allocate/deallocate tiered memory.

API	Description
`void* allocate_TM(size, flags)`	Allocates tiered memory based on flags
`void free_TM(void *ptr)`	Releases tiered memory

C. System Design

In this section, we describe the details of our proposed runtime to reduce the total execution and access latency for containerized workflows with tiered memory-aware page allocation, replacement, and movement policies.

1) Tiered Memory Manager*:* The *Tiered Memory Manager* is the main component of our runtime, that handles coordination between components of our proposed runtime by using a manager and a client deployed on the cluster nodes. The main responsibilities of *Tiered Memory Manager* are: 1) identify various memory types; 2) categorize memory into tiers; 3) create staging buffers on each tier; 4) dynamically adjust buffers based on utilization; and 5) track the hotness/coldness of workflow pages. The *Tiered Memory Manager* identifies various memory types available on the HPC systems and classifies them into tiers with the primary tier being the DRAM memory. The classification of memory into tiers depends on the available memory capacity, access latency, maximum attainable bandwidth, and the interconnect type. It also creates staging buffers on each tier based on the fair-share approach, tier characteristics, and available memory. These buffers are dynamically adjusted based on the memory utilization on each tier and the workflow requirements. Moreover, staging buffers required for transparent data movement across memory tiers are created for each compute node. Lastly, *Tiered Memory Manager* also tracks the hotness of each page of the workflows. The heatmaps are used to identify frequently accessed pages and least frequently accessed pages for efficient page movement between the memory tiers.

The memory allocation, deallocation, and management are done transparently by the runtime based on the workflow requirements and the memory access patterns of the given application. The *Tiered Memory Manager* exposes APIs, shown in Table I, that can be used by workflows to request tiered

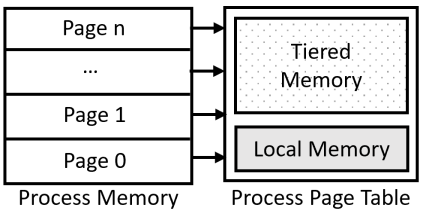

Fig. 3: Tiered memory layout for HPC workflows.

memory for expansion, staging input data, or storing intermediate and output data beyond the initial memory allocation. These APIs are used to allocate and deallocate memory from a specific tier by setting the appropriate flag and for creating shared memory regions between workflows. For example, HPC workflows can use the APIs to request memory from the PMem tier to store data structures that need to be retained. Similarly, for frequently accessed data memory from the CXL tier can be requested to store the prefetched data for caching purposes. The APIs are designed for seamless integration, allowing them to be incorporated into the existing workflow code with minimal modifications.

Once a request is received, the *Tiered Memory Manager* services the request by identifying the ideal memory tier and returning the address space. The requested memory size is in bytes and the flag accepts a combination of the following values: LAT, BW, CAP, SHL which represent latency-sensitive, bandwidth-intensive, capacity, and short-lived, respectively. The LAT flag represents memory that is extremely sensitive to access latency and the page placement necessitates the use of the fastest memory tier. Similarly, the BW flag represents a memory access pattern that requires the highest access bandwidth from either a single or multiple memory tiers. The CAP flag represents memory that is not susceptible to access latency or bandwidth and is primarily use to store pages that are not actively accessed. Lastly, the SHL flag represents memory that is shared between multiple workflows. The flags passed through these APIs allow the user to pass hints regarding the memory resource requirement of workflows. However, these flags are purely advisory and are not mandatory for successful execution. If no flags are provided, then the *Tiered Memory Manager* assigns either single or multiple flags to each workflow based on the previous execution logs, heuristics, and predictor [38]. The *Tiered Memory Manager* also monitors page access patterns and uses this monitoring data for efficient memory allocation and moving data across memory tiers.

Figure 3 shows the layout of the tiered memory system, which consists of CXL-based memory and PMem resources and can span across a cluster of servers. *Tiered Memory Manager* handles all memory access from each workflow to the tiered memory and keeps track of the memory allocations to workflows. It also monitors the memory allocations on each server and dynamically adjusts the memory allocation based on the current memory utilization on each server.

2) Page Allocation Policy: By default, memory is allocated for workflows from the local system memory to maximize performance and reduce the total execution time. However, memory pages are excessively swapped to the slower tiers, e.g., swap space, when the system memory runs out which degrades the performance of running jobs [50], [51]. Our page allocation policy maximizes job performance and reduces the impact of swapping to slower tiers by efficiently utilizing tiered memory and by considering workflow characteristics and the execution sequence. Similarly, it also handles the allocation of additional memory from the tiered memory once the DRAM memory runs out of available space. The policy ensures that the additional memory allocated from the tiered memory has minimal overhead and takes into account the latency requirements of HPC workflows.

Algorithm 1: Page Allocation Policy for HPC workflows with Tiered Memory.

Input : w_id: unique workflow identifier, s: requested mem size, f: list of flags to denote mem characteristics (LAT: lat-sensitive, BW: bw-intensive, CAP: cap-intensive, SHL: short-lived), $alloc\_map$: global mem alloc maps for all workflows, ev: global map of evictable mem available on each tier-($local$, $pmem$, and cxl)

Output: A: memory allocation plan containing a map of memory allocation required from each memory resource

Function `TierAlloc`$(w\_id, s, \langle f \rangle)$**:**
  if $f == NULL$ **then**
    $f = predict\_flags(w\_id, s)$
  if $type(f) == list$ **then**
    $f_{first} = f.pop()$
    $size_{f_{first}} = predict\_flag\_mem\_size(f_{first}, w\_id)$
    `TierAlloc`$(w\_id, size_{f_{first}}, f_{first})$
    `TierAlloc`$(w\_id, s - size_{f_{first}}, f)$
  $A \leftarrow \langle local : 0, pmem : 0, cxl : 0 \rangle$
  if $alloc\_map.find(w\_id)$ **then**
    $A \leftarrow alloc\_map[w\_id]$ `// Find prev. alloc`
  end
  $m = A[local] + A[pmem] + A[cxl]$ `// Alloc'd memory`
  while $m < s$ **do**
    `// Prioritize local memory for lat-sensitive and short-lived tasks`
    if $f == LAT$ ***or*** $f == SHL$ **then**
      if $ev[local] > 0$ **then**
        $A[local] += \min(s - m, ev[local])$
      else if $ev(pmem) > 0$ **then**
        $A[pmem] += \min(s - m, ev[pmem])$
      else if $m < s$ **then**
        $A[cxl] += s - m$ `// Unlimited CXL mem`
    `// Tiered memory allocation for high-bw`
    else if $f == BW$ **then**
      $r \leftarrow 0$ `// Remainder for the next tier`
      for $tier \in [local, pmem, cxl]$ **do**
        $frac[tier] = r + s \times (BW[tier]/BW[total])$
        $curr\_max = \min(frac[tier], ev[tier])$
        $A[tier] += curr\_max$
        $r = curr\_max - frac[tier]$
      end
    `// Addn. memory capacity through CXL`
    else if $f == CAP$ **then**
      $A[cxl] += s - m$
    $m = A[local] + A[pmem] + A[cxl]$
  end
  $alloc\_map \leftarrow alloc\_map \cup A$
  $update\_evictable(A)$
  return A

Our proposed page allocation policy is shown in Algorithm 1. It takes workflow attributes as input, which includes a unique workflow identifier (w_id), the size of the requested

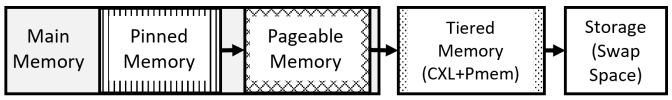

Fig. 4: Memory allocation and page movement for workflows.

memory (s), and an optional list of flags regarding the memory characteristics of the workflow (f). We predict the amount of memory required for each flag using previous execution logs, heuristics, and existing memory predictors [38], [52]. Specifically, the heuristics generate page temperatures by analyzing the page access frequency on each memory tier. For instance, if a job allocates 40 GB of memory and only 512 MB of pages are accessed 80% of the time during the first 20 seconds of execution, then 512 MB of memory is determined to be latency-sensitive (LAT) while the remaining memory is classified as capacity-sensitive (CAP) for the first 20 seconds of execution. To look up execution logs, we utilize workflow configuration information, parameters, flags, etc. For cases where logs are not available or the exact match is not found, we utilize the nearest match as hints for the predictor.

Once the flags are recursively decomposed in atomic values with their corresponding sizes, the current memory allocation of the function on each memory tier (Lines 9-12) is fetched. The total size required is updated for the given function based on previous allocations (Line 13). Next, we iteratively allocate suitable memory pages from each memory tier based on the function requirements (Lines 14-33). For latency-sensitive (LAT) and short-lived (SHL) workflows, the policy attempts greedy allocation of memory starting from the fastest to the slowest tier in a cascading fashion (Lines 15-21). This approach mitigates the challenge of higher access latency for such workflows. Part of the memory belonging to the latency-sensitive and short-lived workflows is pinned to guarantee the required performance and the remaining portion is tagged as a pageable region that can be used for swapping and replacement as shown in Figure 4. For simplicity, our policy assumes that an unlimited memory is available over the CXL interconnect and the remaining memory can be directly allocated from CXL. Although such greedy-based decomposition leads to suboptimal initial allocations for workflows that are launched at a later time by forcing them to allocate memory from high-latency slower memory tiers (e.g., CXL), our page replacement algorithm (discussed in Section III-C3) effectively mitigates this overhead. For latency-sensitive workflows, our runtime pre-faults [53] the memory addresses to reduce the overhead of page faults during memory access.

For bandwidth-intensive workflows (BW), we use a multi-path memory access approach that allocates memory on each available tier ($local$, $pmem$, cxl) to provide maximum available bandwidth to the workflow. The memory allocated on each tier is directly proportional to the available read/write throughput observed from that tier. For cases where faster memory tiers experience higher contention levels, and the required memory is not available (Line 26), only partial memory is allocated (Line 27), and the remainder of memory from the next fastest memory tier (Line 28). Finally, for capacity-intensive (CAP) workflows, the entire memory is allocated directly from the CXL memory tier. Finally, based on the amount of memory allocated on each tier, the corresponding allocation entry in the global allocation and eviction maps are updated (Lines 34-35) and the memory allocation plan A is returned.

Algorithm 2: Page Replacement Policy to manage hot/cold pages across multiple memory tiers.

Input : r: Number of pages to replace from memory
Output: *None*

```
1  begin
2      t ← 0                          // Number of replaced pages
3      while t < r do
4          victim_pages ← lru_pagable(r − t)
5          victim_pages ← remove_lat_or_shl(victim_pages)
6          t+ = victim_pages
7          move_out(victim_pages)
8          update_pg_table(victim_pages)
9      end
10     update_alloc_map(r)
11 end
```

We note that the algorithmic complexity of the proposed page allocation policy is a linear function of the number of memory tiers. However, since we consider the case of only three memory tiers, the complexity becomes constant $\mathcal{O}(1)$. Such low complexity is particularly important for time-sensitive HPC workflows.

3) Page Replacement Policy: Many page replacement techniques have been extensively studied for conventional memory subsystems [54]–[57] to create space in DRAM for pages that have been swapped out to slower storage tiers. The default behavior of the Linux kernel is to select a set of candidate pages based on various heuristics [54], such as least-recently-used, most-recently-used, and optimal page replacement, that can be evicted to a disk-based swap partition to be replaced with the requested page. However, this approach is agnostic to the underlying heterogeneous memory tiers and results in sub-optimal page replacements to slower disk-based storage tiers, leading to resource underutilization, performance degradation due to major page faults, and low system throughput.

To address the above challenges, we propose a page replacement policy, shown in Algorithm 2, to mitigate the impact of suboptimal page faults to accommodate bandwidth-intensive and time-critical HPC workflows. We adopt a dynamic memory eviction model based on the characteristics of the function such as latency-sensitivity or short-lived function. Our page replacement does not depend on the input flags or predictor output, however, these flags enable fine-tuned page replacement for specific workflow types. Note that the predictor is only used for estimating initial allocation using previous execution logs or heuristics in the absence of flags. The replacement policy also considers page temperatures and memory access patterns for all colocated workflows to identify and prioritize the eviction of cold pages. The algorithm takes the number of pages to be replaced (r) as input based on the system-level page faults and filters out the memory pages belonging to the above class of applications (Lines 4-5) based on the victim pages identified by the Linux kernel. The

filtered pages are tracked and moved to the lower memory tier rather than swapped out to the underlying disk-based swap space (Line 7). Once the victim pages are identified, they are swapped to the swap space and replaced with the requested page by the application. Finally, the allocation map is updated with the replaced pages (Line 8). Our page replacement policy ensures that the memory pages belonging to the latency-sensitive and short-lived workflows are not blindly swapped out by the Linux kernel resulting in major page faults that eventually degrade application performance.

4) Intelligent Page Movement Policy: To improve application performance and reduce the latency of accessing memory pages, we propose an intelligent page movement policy that proactively moves memory pages between various memory tiers and implements a proactive page-swapping mechanism that swaps out memory pages to the CXL memory. To mitigate the negative impacts of proactive swapping, the swapped-out memory pages are cached in the page cache if there is enough memory available on the main memory and are marked as dispensable and the corresponding page table entry is updated. If enough system memory is not available, then the memory pages are simply moved to the CXL memory tier. Once the system memory runs out, instead of swapping pages to the swap space, the pages in the page cache are first swapped out and then the workflow memory pages are swapped. The page movement from the main memory is based on workflow characteristics, e.g., latency-sensitivity, to the CXL memory and then eventually to the local disk. The proactive page swapping also performs memory compaction to reduce fragmentation and enable contiguous memory blocks to be allocated to workflows for colocating more workflows on the system, thus improving system utilization.

The proposed page movement policy also moves pages between persistent and CXL-attached memory tier based on the available page access heatmaps as discussed in Secion III-C1. This enables the runtime to effectively move pages to faster memory tiers that were previously identified as cold but later categorized as hot pages. Our application-aware intelligent page movement policy prioritizes application pages that do not belong to latency-sensitive or short-lived applications. If a page belonging to the above classes of applications must be moved, then the policy prioritizes pages belonging to the pageable memory region as defined in the page allocation map. Our intelligent page movement policy minimizes the impact of page swapping by enabling the swapped pages to be available in the fastest available memory tier. Finally, our page movement policy reduces the number of major page faults and subsequently increases the number of minor page faults as the page is accessible on other memory tiers or the page cache.

5) Management of Shared Memory Across Workflows: CXL memory provides a fast backend to improve the performance of shared memory regions for HPC workflows. Input or read-only data shared between workflows can be staged in the CXL memory, which can be leveraged by the HPC job scheduler e.g. SLURM, to launch workflows at scale and minimize the scale-up time and data transfers between workflows. For example, launching thousands of HPC workflows using a custom Singularity container image requires the image to be moved to all the servers that will run the job workflows. This creates a network and I/O bottleneck when a large number of workflows access the same data resulting in an increased execution time to prepare the runtime and increase the cold-start latency for containers. For simplicity, we assume that the workflow manages the shared memory and handles locking mechanisms as offered by several libraries [58], [59] to block read or write operations during an ongoing write to the shared memory region. We provide three strategies for efficiently managing shared memory between workflows at the workflow and platform levels. First, shared memory pages are made locality-aware by incorporating the location of workflows accessing the shared memory by the HPC job scheduler. Such memory pages are hosted on the CXL memory accessible to both workflows, and the memory pages are cached in the local buffers for fast access on each server. Second, to improve the capability of the HPC job scheduler to scale up workflows and reduce the cold start latency, we leverage the CXL memory to host container images and application data. Third, our proposed runtime keeps track of the memory tagged as shared memory and ensures that during a scale-down event, the shared memory is not deallocated. The shared memory is freed when all references in the corresponding page tables have been removed. These approaches ensure that the shared memory is effectively allocated, managed, and utilized for large-scale containerized HPC workflow deployments.

IV. Performance Evaluation

In this section, we present the evaluation of the proposed memory management policies for HPC workflows using tiered memory. We explain our prototype implementation, evaluation methodology, testbed, workflows, and performance metrics that we use to analyze and compare our proposed runtime with baseline and other alternative execution approaches.

A. Prototype Implementation

We implemented the proposed runtime using approximately 1500 lines of C code including two Linux Kernel modules, and integrated it with the HPC job scheduler SLURM [60], container framework Singularity [10], and Pegasus [28] WMS. Our runtime allocates the initial memory and serves workflow requests using the provided APIs for allocating and deallocating memory across the tiered memory. In our prototype implementation, we modify SLURM to support the required flags along with the job script to infer hints about the characteristics of workflow for allocating and deallocating memory for that workflow. Our hand-tuned implementation customizes the page allocation and replacement policies to incorporate additional memory tiers hosting one workflow per container and launching multiple workflows on the HPC cluster.

B. Evaluation Methodology

We compare our runtime with the baseline scenario where HPC workflows are colocated and frequently run out of

memory resulting in swapping out of memory pages. We also compare its performance with a more realistic scenario where workflows memory is allocated from CXL memory without considering the workflow performance characteristics. In our evaluation, we study the following metrics to demonstrate the effectiveness of our proposed approach: total workflow execution time, number of page faults, total execution makespan of HPC workflows submitted as batch jobs, and workflow and cluster scalability. The total execution time is the time required to complete the scheduled workflows and return the results. The bandwidth and latency numbers are reported for the CXL memory allocated to the workflows and compared to the local memory and swap space. Lastly, we use the number of memory accesses, the amount of data swapped to disk, and CXL memory to gauge the performance of the memory management policies. To evaluate workflows that have varying memory access patterns we randomly select workflows and substitute them with versions that request additional memory during execution using our APIs and incorporating specific flags. This approach ensures that the experimentation environment remains dynamic, facilitating the exploration of various memory access patterns that may evolve during execution. We run each experiment 10 times and report the average. Overall, we observe a negligible variance, i.e., less than 5% between different executions of the same experiment in our evaluation.

C. Evaluation Setup

1) Testbed: Our evaluation setup consists of a cluster of 8 bare-metal servers connected using 10G Ethernet. Each server has two Intel Xeon Gold 6126/6240R/6242 processors, contains 512 GB of main memory, 1 TB of Intel Optane DC persistent memory, and runs Ubuntu 22.04 LTS server operating system. We deploy SLURM along with Singularity on all servers in our evaluation setup. We provision the tiered memory using the local DRAM, persistent, and CXL memory available on the servers via the CXL interconnection. The CXL memory is emulated [40], [61], [62] using the remote NUMA socket as advocated by POND [63] and CXLMemSim [62]. In our testbed, we observe the local and remote NUMA latencies to be ∼80 ns and ∼140 ns, respectively, which represent the approximate latency of a CXL-attached memory [46], [63].

2) Evalation Workflows: Modern HPC workflows [64]–[67] typically consist of core scientific computing (SC) simulations [5], surrogate deep-learning (DL) tasks that assist the core simulation [68]–[70], data compression/decompression (DC) [71]–[73] for collective communications and storage, and data mining (DM) [74]–[76] required by analytics engines to steer the experimental trajectory in real-time. In our evaluations, we consider HPC workflows composed of these where each workflow represents jobs with unique characteristics, i.e., computing (requires powerful CPUs), data (processes large volumes of data), bandwidth-intensive (requires large bandwidth), latency-sensitive (requires fast access), and short-lived. DL is a data and bandwidth-intensive workflow in which we train the popular NLP model, i.e., Bert [43], over the IMDB dataset [77] for a total of 5 epochs. The DM workflow

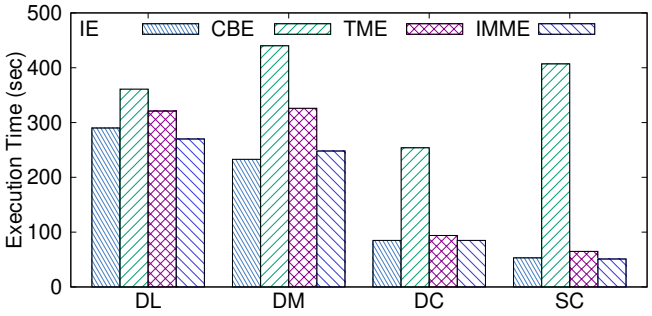

Fig. 5: Impact of our runtime on the studied execution environments.

is a latency-sensitive workflow running a task on Spark that performs ETL [78] over the US census data [79] and computes the diversity index. The DC workflow is a compute and data-intensive workflow in which we run Zip [80] compression on a set of 50 GB input files. The SC workflow runs BFS using igraph [81] on a binary tree.

3) Execution Environments: To study the impact of our memory management policies, we define four realistic execution environments for running HPC workflows based on the availability of memory and storage subsystems. These execution environments are:

1) ***Ideal Environment*** **(IE)** represents an ideal baseline environment with enough local memory.
2) ***Constrained Baseline Environment*** **(CBE)** represents a more realistic environment with limited system memory and memory pages are frequently swapped out.
3) ***Tiered Memory Environment*** **(TME)** is based on the *Constrained Baseline Environment* but uses tiered memory for memory allocation with default Linux page promotion and demotion based on page temperatures.
4) ***Intelligent Memory Management Environment*** **(IMME)** is based on the *Tiered Memory Environment* and uses our intelligent memory management policies.

D. Performance Results

In this section, we present the performance results of our proposed approaches by executing the workflows on the studied execution environments and comparing their performance.

1) Impact of Tiered Memory on Total Execution Time of HPC Workflows: We study the impact of allocating tiered memory to HPC workflows and report the total execution time for the studied execution environments. The results are shown in Figure 5. We observe that the *Ideal Environment* takes the least execution time for all studied workflows because sufficient system memory is available to host the entire footprint of HPC workflows in memory. We observe degraded performance for the *Constrained Baseline Environment* as compared to *Ideal Environment* due to the limited system memory availability and frequent swapping of workflow memory pages to slower tiers. Similarly, the performance of latency-sensitive and short-lived, i.e., the DM workflows, drops significantly due excessive swapping and contention. However, the availability of tiered memory in the *Tiered Memory Environment* reduces this impact by providing a faster alternative and performs

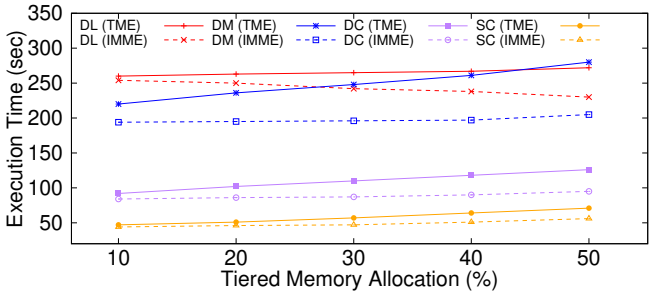

Fig. 6: Impact of our proposed runtime on the workflow performance with varying tiered memory availability.

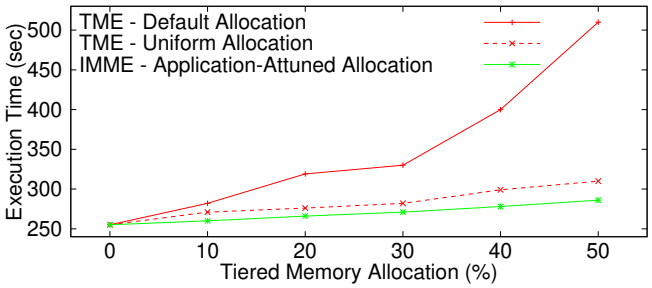

Fig. 7: Impact of our memory allocation policy on execution time.

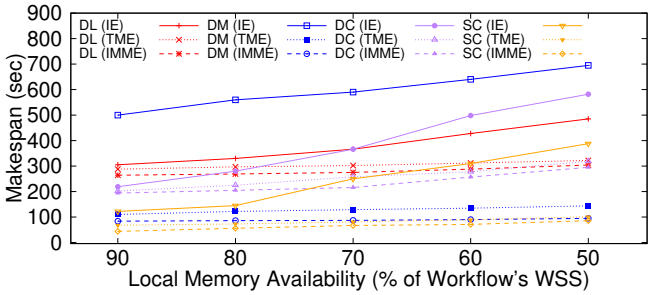

Fig. 8: Impact of our memory allocation policy on the execution makespan of the studied workflows.

better than the *Constrained Baseline Environment*. Similarly, for *Intelligent Memory Management Environment*, we observe that our runtime utilizes tiered memory to improve the performance of workflows by allocating memory to appropriate workflows, intelligently moving pages between memory tiers, and proactive swapping memory pages to the CXL memory tier. Overall, we observe that the *Intelligent Memory Management Environment* reduces the execution time of studied workflows by up to 7%, 87%, and 25% as compared to the *Ideal Environment*, *Constrained Baseline Environment*, and *Tiered Memory Environment*, respectively.

We also study the impact of varying tiered memory allocations on the execution time. Figure 6 shows the results. Here, we vary the tiered memory allocation from 10% to 50%, where each data point represents the percentage of workflow memory allocated from the CXL memory tier. In the *Tiered Memory Environment*, we observe that as we increase the allocation of CXL memory to the workflows, the execution time increases due to the additional latency associated with accessing the CXL memory. We also observe that the *Tiered Memory Environment* does not manage tiered memory efficiently and causes bandwidth-intensive workflows to not fully utilize the additional available bandwidth, and latency-sensitive workflows to experience additional latency over the CXL interconnect. Since our proposed runtime allocates tiered memory based on workflow requirements and characteristics, we observe a reduced execution time for the studied workflows. Moreover, workflows that require additional memory continue to execute by expanding their memory footprint on the tiered memory which would otherwise crash due to limited local memory or fixed memory allocations. Overall, we observe that our memory management policies improve workflow performance by up to 80% as compared to the *Tiered Memory Environment* by efficiently allocating and managing memory tiers based on workflow characteristics and requirements.

2) Impact of Page Allocation Policy on Workflow Performance: We study the impact of our page allocation policy on workflow performance by launching multiple instances of the studied workflows on the HPC cluster. To evaluate the effectiveness of our allocation policy, we report the total execution time of each workflow in Figure 7. We compare our page allocation policy with two approaches: 1) the *Default Allocation* policy where the system memory and CXL memory are allocated to workflows regardless of its requirements; 2) the *Uniform Allocation* policy allocates CXL memory to all workflows in a uniform fashion regardless of the workflow requirements. We observe that the *Default Allocation* policy allocates CXL memory to workflows based on its demand without catering to the class it belongs to and results in degraded performance for latency-sensitive and short-lived workflows. This approach is beneficial for latency-sensitive workflows and capacity-intensive workflows, but the performance of latency-sensitive workflows degrades as soon as the memory footprint overflows to tiered memory. The *Uniform Allocation* policy results in the worst performance for latency-sensitive workflows as they experience additional access latency of the tiered memory due to interleaving. However, interleaving results in improved performance for bandwidth-intensive workflows due to the availability of additional bandwidth. Overall, the *Uniform Allocation* outperforms the *Default Allocation*, however, the memory allocation is not aware of the workflow characteristics. The performance of *Uniform Allocation* can be further improved with weighted interleaving, however, setting weights does not consider the characteristic for all workflow types. We also observe that our memory allocation policy reduces the total workflow execution time by intelligently allocating CXL memory to workflows to minimize the impact of additional access latency. Overall, we observe that our allocation policy reduces the execution time by 44% and 8% on average as compared to the *Default Allocation* and *Uniform Allocation* strategies, respectively.

We also study the impact of our memory allocation policy on each class of workflow by varying the percentage of available DRAM to each workflow as a function of its working set size (WSS). The results are shown in Figure 8. We observe

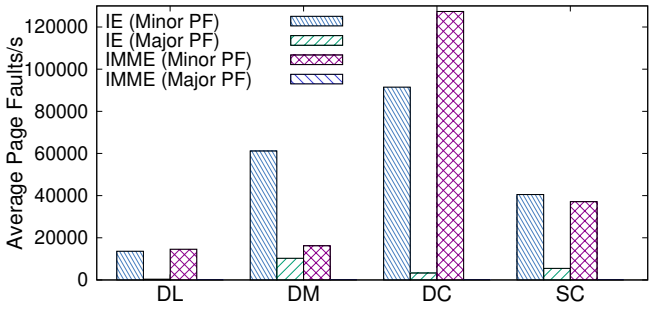

Fig. 9: Impact of our page movement policy on workflow page faults.

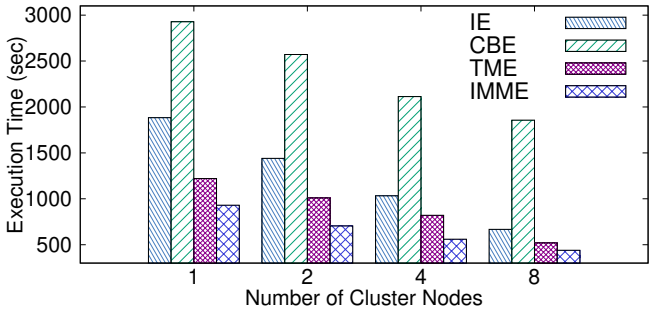

Fig. 10: Impact of our runtime on execution time of 3000 workflows on an 8-node cluster.

that as the amount of DRAM available to latency-sensitive workflows decreases, the memory access time increases resulting in a significant impact on makespan and performance. Similarly, for bandwidth-intensive workflows, we observe that our memory allocation policy leverages the available CXL memory to improve the overall throughput by leveraging the additional memory tiers. For *Tiered Memory Environment*, we observe that as the memory available to workflows decreases, the hot pages are promoted to DRAM reducing the impact of additional latency of CXL memory. Moreover, the speedup is achieved as the additional memory availability reduces the impact of swapping to slower storage for the *Ideal Environment*. Moreover, workflows that require large memory capacity to successfully execute, benefit from potentially unlimited memory availability from the CXL memory. Overall, we observe that our memory allocation policy reduces the overall makespan by 25%, 85%, 35%, and 71% on average compared to *Ideal Environment* for deep learning, data mining, data compression, and scientific workflows, respectively. Similarly, we observe that our memory allocation policy reduces the overall makespan by 8%, 31%, 9%, and 22% on average compared to *Tiered Memory Environment* for deep learning, data mining, data compression, and scientific workflows, respectively.

3) Impact of Page Movement Policy on Workflow Performance: We study the impact of our intelligent page movement policy by observing the page fault statistics for the studied workflows. The results are shown in Figure 9. We observe that in the *Ideal Environment*, the Linux kernel swaps out memory pages based on the least recently used (LRU) policy regardless of the workflow requirements or characteristics. This causes a performance drop in latency-sensitive workflows which are most susceptible to additional latency when pages are swapped back in by the Linux kernel. We observe that with the availability of CXL memory, our page movement policy reduces the number of pages that are swapped to the disk by reducing the major page faults, thereby, improving workflow performance. However, workflows that are extremely sensitive to latency suffer additional latency when reading and writing from CXL memory. Our intelligent page movement policy reduces the number of major page faults by moving pages to the CXL memory which in turn increases minor page faults for each workflow. Furthermore, Linux swapping increases workflow execution time even with CXL memory. We observe that our intelligent page movement policy performs workflow-attuned page movement and ensures that the memory pages are available in the fastest tier and pages of latency-sensitive and short-lived workflows are protected from swapping. Our intelligent memory movement also performs proactive swapping in the background in addition to moving memory pages between various memory tiers. Our proactive swapping moves out workflow memory pages that are less sensitive to the overhead of moving pages back into the memory. This enables keeping more pages of latency-sensitive and short-lived workflows in the memory. Overall, we observe that our workflow-attuned page movement and proactive page-swapping improve workflow performance by 46% as compared to the default swapping policy.

4) Scalability Analysis of our Proposed Runtime on Workflow Performance: We increase the size of the HPC cluster and the number of concurrent workflows to study the impact of our proposed runtime on a large HPC cluster. We launch 2000 instances of the studied workflows (150 for DL, 1100 for DM, 150 for DC, 600 for SC workflows) concurrently and observe the impact on the workflow execution time. Figure 10 shows the results of this experiment. We observe that the execution time is significantly reduced with the increasing number of cluster nodes thanks to the overall memory allocation and page movement on each server leveraging the CXL memory effectively. With the *Constrained Baseline Environment*, the execution time is the highest due to the limited resource availability and the contention at each node of the cluster. As the memory utilization of the system increases due to colocation, the *Tiered Memory Environment* efficiently utilizes the tiered memory to promote hot pages to faster tiers improving the overall workflow performance. Moreover, we observe that for large-scale invocations, the overall execution time and the workflow startup time are reduced with *Intelligent Memory Management Environment* due to the effective placement of shared files on the CXL memory that is accessible to all the nodes in the cluster. Overall, we observe a performance improvement of up to 51%, 76%, and 32% compared to the *Ideal Environment*, *Constrained Baseline Environment*, and *Tiered Memory Environment*, respectively.

We also study the impact of concurrent workflow invocations on the overall execution time of batch HPC jobs containing all studied workflows with varying, i.e., 100, 200, 400, and 800, instances. The results are shown in Figure 11.

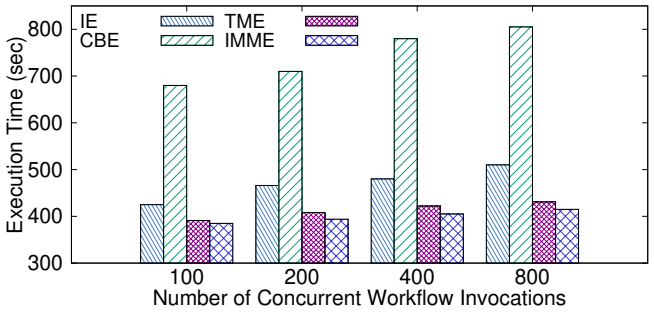

Fig. 11: Impact of our runtime on execution time on an 8-node cluster.

We observe that as the number of concurrent workflows increases, the execution time also increases due to resource contention at servers. We observe a negligible overhead, i.e., 4%, of our proposed runtime as the workflows are scaled up due to efficient multi-tiered memory allocation policy and intelligent page movement to ensure that the workflow startup time is reduced. Overall, we observe that our proposed runtime reduces the execution time by up to 19%, 48%, and 4% compared to the *Ideal Environment*, *Constrained Baseline Environment*, and *Tiered Memory Environment*, respectively.

V. Related Work

Workflow Management Systems and HPC Workflows: Modern large-scale HPC workloads typically consist of multiple complex workflows represented as DAG, and executed on HPC systems. To address complex data, task, and resource dependencies, workflow management systems (WMS) are used to run such HPC workloads. WMSs such as Balsam [32], Pegasus [28], and Cromwell [29] have been extensively used for steering complex scientific experiments [82], real-time data analysis [83], ensembles [84], deep-learning based surrogates [85], etc. for both baremetal and containerized executions. WMSs utilize HPC schedulers such as SLURM [60] or COBALT [86] to handle scheduling requests. However, none of the WMSs perform task scheduling based on the memory requirement of the workflow/tasks, resulting in suboptimal resource allocation, utilization, and reduced performance.

Tiered Memory Systems: Tiered memory systems in HPC address the increasing memory capacity, bandwidth, and latency requirements of HPC workflows. These systems leverage different memory technologies, e.g., DRAM, PMem, and CXL-based memory, where each memory type offers distinct performance characteristics [87], [88]. DRAM provides high-speed and low-latency access while PMem offers non-volatile memory and bridges the gap between DRAM and storage, enabling data persistence even during power loss [89]–[91]. CXL memory provides fast, high-capacity, and low-latency access to applications enhancing scalability and resource pooling in tiered memory systems for improved HPC performance [41], [92]–[94]. Tiered memory systems also improve overall memory utilization by intelligently allocating data to the most appropriate tier based on access patterns and performance requirements [95]–[97]. However, neither the applications nor the platforms are optimized to leverage the true potential of tiered memory systems resulting in degraded application performance and system utilization.

Memory Management Approaches: Tiered memory management approaches have been extensively explored by several studies such as Nimble [57], TPP [98], HeMem [99], Pond [41], AutoTM [100] etc. These approaches perform application-agnostic memory allocations and page movement across various memory tiers. However, these techniques result in degraded performance for colocated HPC workflows with diverse memory requirements. Moreover, they perform strictly hierarchical page movement and do not perform concurrent tiered memory allocation to optimize bandwidth through parallel interconnects. Other efforts [101]–[105] either solve the challenge of memory management for terabyte-scale applications (e.g., HM-Keeper [101]) or partially optimize and automate memory management across multiple memory tiers. Similarly, MTM [106] performs application-transparent page management based on profiling, multi-tiered page migration policy, and huge page awareness. Our approach extends on the general design ideas of the above state-of-the-art tiered memory approaches, and incorporates applications' memory characteristics for efficient memory management.

VI. Conclusion

Containerized HPC workflows have gained rapid adoption for running HPC workloads due to their native support for high concurrency and scalability. However, running multiple containerized HPC workflows present unique challenges associated with memory including performance penalties due to limited memory bandwidth, latency, and workflow-agnostic page management. Recently, tiered memory systems have been explored to address the above challenges, however, current memory management approaches do not perform application-attuned memory allocation and management to maximize workflow performance. In this paper, we explore tiered memory systems for running containerized HPC workflows and propose application-attuned intelligent page allocation, movement, and replacement policies to improve performance. We integrate our proposed runtime with popular HPC scheduler (SLURM) and container runtime (Singularity) and evaluate its performance using diverse HPC workflows with various computing, capacity, bandwidth, and latency requirements. Our evaluation shows that our proposed runtime reduces workflow execution times by up to 51%, 87%, and 35% as compared to the ideal, realistic, and optimized tiered execution environments, respectively. In the future, we plan to extend our page allocation policy to support variable latency and bandwidth to enable more efficient page replacement and movement. Furthermore, we plan on extending our implementation to include accelerator memory.

Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CSR-2315851/2106634/2106635/2312785/CCF-1919113/OAC-2004751, Sony Faculty Innovation Award (Contract AG3ZURVF), and Cisco Research Award (Contract 878201).

REFERENCES

[1] Alexander Brace, Igor Yakushin, Heng Ma, Anda Trifan, Todd Munson, Ian Foster, Arvind Ramanathan, Hyungro Lee, Matteo Turilli, and Shantenu Jha. Coupling streaming ai and hpc ensembles to achieve 100–1000× faster biomolecular simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 806–816. IEEE, 2022.

[2] William D Marks, Jun Yokose, Takashi Kitamura, and Sachie K Ogawa. Neuronal ensembles organize activity to generate contextual memory. *Frontiers in Behavioral Neuroscience*, 16:75, 2022.

[3] Sam Partee, Matthew Ellis, Alessandro Rigazzi, Andrew E Shao, Scott Bachman, Gustavo Marques, and Benjamin Robbins. Using machine learning at scale in numerical simulations with smartsim: An application to ocean climate modeling. *Journal of Computational Science*, 62:101707, 2022.

[4] Kazuki Maeda, Thiago Teixeira, Jonathan M Wang, Jeffrery Hokanson, Caetano Melone, Mario Di Renzo, Steve Jones, Javier Urzay, and Gianluca Iaccarino. An integrated heterogeneous computing framework for ensemble simulations of laser-induced ignition. In *AIAA AVIATION 2023 Forum*, page 3597, 2023.

[5] J Luc Peterson, K Athey, PT Bremer, V Castillo, F Di Natale, JE Field, D Fox, J Gaffney, D Hysom, SA Jacobs, et al. Merlin: enabling machine learning-ready hpc ensembles. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019.

[6] Logan Ward, Ganesh Sivaraman, J Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C Redfern, Rajeev S Assary, Kyle Chard, Larry A Curtiss, et al. Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments*, pages 9–20. IEEE, 2021.

[7] Yinzhi Wang, R. Todd Evans, and Lei Huang. Performant container support for hpc applications. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Subil Abraham, Arnab K. Paul, Redwan Ibne Seraj Khan, and Ali R. Butt. On the use of containers in high performance computing environments. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 284–293, 2020.

[9] Moiz Arif, Kevin Assogba, and M. Mustafa Rafique. Canary: Fault-tolerant faas for stateful time-sensitive applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[10] David Godlove. Singularity: Simple, secure containers for compute-driven workloads. In *Proc. of the Practice and Experience in Advanced Research Computing on Rise of the Machines*, pages 1–4. 2019.

[11] Riken Center for Computational Science. About Fugaku. https://www.r-ccs.riken.jp/en/fugaku/about/. Accessed on: 09-29-2023.

[12] Kim Askey. Deep Learning to Predict Protein Functions at Genome Scale – OLCF. https://www.olcf.ornl.gov/2022/01/10/scientists-use-summit-supercomputer-deep-learning-to-predict-protein-functions-at-genome-scale/. Accessed on: 04-06-2023.

[13] Taylor Childers. Singularity on Theta: How to Build and Scale Containers at the ALCF — Argonne Leadership Computing Facility. https://www.alcf.anl.gov/support-center/training-assets/singularity-theta-how-build-and-scale-containers-alcf. Accessed: 2023-04-07.

[14] Argonne National Laboratory. Containers — ALCF User guides. https://docs.alcf.anl.gov/polaris/data-science-workflows/containers/containers/. Accessed on: 04-06-2023.

[15] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I. Ingólfsson, Joseph Koning, Tapasya Patki, Thomas R.W. Scogland, Becky Springmeyer, and Michela Taufer. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems*, 110:202–213, 2020.

[16] Tu Mai Anh Do, Loïc Pottier, Rafael Ferreira da Silva, Frédéric Suter, Silvina Caíno-Lores, Michela Taufer, and Ewa Deelman. Co-scheduling ensembles of in situ workflows. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 43–51, 2022.

[17] Tu Mai Anh Do, Loïc Pottier, Rafael Ferreira da Silva, Silvina Caíno-Lores, Michela Taufer, and Ewa Deelman. Performance assessment of ensembles of in situ workflows under resource constraints. *Concurrency and Computation: Practice and Experience*, page e7111.

[18] Stephen Herbein, Ayush Dusia, Aaron Landwehr, Sean McDaniel, Jose Monsalve, Yang Yang, Seetharami R Seelam, and Michela Taufer. Resource management for running hpc applications in container clouds. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 261–278. Springer, 2016.

[19] U.S. DEPARTMENT OF ENERGY. U.S. Department of Energy and Intel to deliver first exascale supercomputer. https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer, 2019.

[20] Barcelona Supercomputing Center. BSC executes, for the first time, big encrypted neural networks using Intel Optane Persistent Memory and Intel Xeon Scalable Processors. https://www.bsc.es/news/bsc-news/bsc-executes-the-first-time-big-encrypted-neural-networks-using-intel-optane-persistent-memory-and, 2021.

[21] University of Tsukuba. Pegasus - Big memory supercomputer. https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#Pegasus, 2023.

[22] Adnan Maruf, Daniel Carlson, Ashikee Ghosh, Manoj Saha, Janki Bhimani, and Raju Rangaswami. Allocation policies matter for hybrid memory systems. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 321–322, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Hwanjun Lee, Seunghak Lee, Yeji Jung, and Daehoon Kim. T-cat: Dynamic cache allocation for tiered memory systems with memory interleaving. *IEEE Computer Architecture Letters*, 22(2):73–76, 2023.

[24] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of DRAM cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 519–534, Boston, MA, July 2023. USENIX Association.

[25] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 11–20, 2022.

[26] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023.

[27] Diego Moura, Daniel Mossé, and Vinicius Petrucci. Performance characterization of autonuma memory tiering on graph analytics. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 171–184, 2022.

[28] Ewa Deelman, Rafael Ferreira da Silva, Karan Vahi, Mats Rynge, Rajiv Mayani, Ryan Tanaka, Wendy Whitcup, and Miron Livny. The pegasus workflow management system: translational computer science in practice. *Journal of Computational Science*, 52:101200, 2021.

[29] Kate Voss, Geraldine Van Der Auwera, and Jeff Gentry. Full-stack genomics pipelining with gatk4 + wdl + cromwell [version 1; not peer reviewed], 2017.

[30] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[31] Rosa Maria Badia Sala, Eduard Ayguadé Parra, and Jesús José Labarta Mancho. Workflows for science: A challenge when facing the convergence of hpc and big data. *Supercomputing frontiers and innovations*, 4(1):27–47, 2017.

[32] Michael A Salim, Thomas D Uram, J Taylor Childers, Prasanna Balaprakash, Venkatram Vishwanath, and Michael E Papka. Balsam: Automated scheduling and execution of dynamic, data-intensive hpc workflows. *arXiv preprint arXiv:1909.08704*, 2019.

[33] Ivy Peng, Ian Karlin, Maya Gokhale, Kathleen Shoga, Matthew Legendre, and Todd Gamblin. A holistic view of memory utilization on hpc systems: Current and future trends. In *The International Symposium on Memory Systems*, pages 1–11, 2021.

[34] Dan Huang, Zhenlu Qin, Qing Liu, Norbert Podhorszki, and Scott Klasky. A comprehensive study of in-memory computing on large hpc systems. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 987–997. IEEE, 2020.

[35] Ranjan Sarpangala Venkatesh, Tony Mason, Pradeep Fernando, Greg Eisenhauer, and Ada Gavrilovska. Scheduling hpc workflows with intel optane persistent memory. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 56–65. IEEE, 2021.

[36] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.

[37] Sebastian Lührs, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings. Flexible and generic workflow management. In *Parallel Computing: On the Road to Exascale*, pages 431–438. IOS Press, 2016.

[38] Kevin Assogba, Moiz Arif, M. Mustafa Rafique, and Dimitrios S. Nikolopoulos. On realizing efficient deep learning using serverless computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 220–229, 2022.

[39] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[40] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. *arXiv preprint arXiv:2211.02682*, 2022.

[41] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[42] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, DaMoN'22, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[44] Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Karim El Tal, and Stefan Tai. An evaluation of faas platforms as a foundation for serverless big data processing. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 1–9, 2019.

[45] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 228–244, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. Exploiting cxl-based memory for distributed deep learning. In *2022 51st International Conference on Parallel Processing*, 2022.

[47] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. Training resilience with persistent memory pooling using cxl technology. In *Heterogeneous and Composable Memory Workshop at HPCA, 2023*. IEEE, 2023.

[48] Connor Imes, Steven Hofmeyr, Dong In D. Kang, and John Paul Walters. A case study and characterization of a many-socket, multi-tier numa hpc platform. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 74–84, 2020.

[49] Daniel S Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D Hill, et al. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 2023.

[50] Xinjian Long, Xiangyang Gong, Bo Zhang, and Huiyang Zhou. An intelligent framework for oversubscription management in cpu-gpu unified memory. *Journal of Grid Computing*, 21(1):11, 2023.

[51] Salman Abdul Baset, Long Wang, and Chunqiang Tang. Towards an understanding of oversubscription in cloud. In *Hot-ICE*, 2012.

[52] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[53] Niall Douglas. User mode memory page allocation: A silver bullet for memory allocation? *arXiv preprint arXiv:1105.1811*, 2011.

[54] Amit S Chavan, Kartik R Nayak, Keval D Vora, Manish D Purohit, and Pramila M Chawan. A comparison of page replacement algorithms. *International Journal of Engineering and Technology*, 3(2):171, 2011.

[55] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, 1999.

[56] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461, 2020.

[57] Zi Yan, Daniel Lustig, David Nellans, and Abhishek 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 '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Tim Blechmann. Boost. lockfree. *Boost C++ Libraries*, 2013.

[59] Dave Dice and Nir Shavit. Tlrw: return of the read-write lock. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 284–293, 2010.

[60] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[61] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, pages 1–5. 2022.

[62] Yiwei Yang, Pooneh Safayenikoo, Jiacheng Ma, Tanvir Ahmed Khan, and Andrew Quinn. Cxlmemsim: A pure software simulated cxl. mem for performance characterization. *arXiv preprint arXiv:2303.06153*, 2023.

[63] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[64] Rafael Ferreira da Silva, Henri Casanova, Kyle Chard, Ilkay Altintas, Rosa M Badia, Bartosz Balis, Tainã Coleman, Frederik Coppens, Frank Di Natale, Bjoern Enders, Thomas Fahringer, Rosa Filgueira, Grigori Fursin, Daniel Garijo, Carole Goble, Dorran Howell, Shantenu Jha, Daniel S. Katz, Daniel Laney, Ulf Leser, Maciej Malawski, Kshitij Mehta, Loïc Pottier, Jonathan Ozik, J. Luc Peterson, Lavanya Ramakrishnan, Stian Soiland-Reyes, Douglas Thain, and Matthew Wolf. A community roadmap for scientific workflows research and development. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 81–90, 2021.

[65] J Luc Peterson, Ben Bay, Joe Koning, Peter Robinson, Jessica Semler, Jeremy White, Rushil Anirudh, Kevin Athey, Peer-Timo Bremer, Francesco Di Natale, et al. Enabling machine learning-ready hpc ensembles with merlin. *Future Generation Computer Systems*, 131:255–268, 2022.

[66] Graphcore. AI for Simulation: How Graphcore is Helping Transform Traditional HPC. https://www.graphcore.ai/posts/ai-for-simulation-how-graphcore-is-helping-transform-traditional-hpc, 2022.

[67] Jakob Lüttgau, Shane Snyder, Philip Carns, Justin M Wozniak, Julian Kunkel, and Thomas Ludwig. Toward understanding i/o behavior in hpc workflows. In *2018 IEEE/ACM 3rd international workshop on parallel data storage & data intensive scalable computing systems (PDSW-DISCS)*, pages 64–75. IEEE, 2018.

[68] Rushil Anirudh, Jayaraman J Thiagarajan, Peer-Timo Bremer, and Brian K Spears. Improved surrogates in inertial confinement fusion with manifold and cycle consistencies. *Proceedings of the National Academy of Sciences*, 117(18):9741–9746, 2020.

[69] Matthew Chantry, Sam Hatfield, Peter Dueben, Inna Polichtchouk, and Tim Palmer. Machine learning emulation of gravity wave drag in numerical weather forecasting. *Journal of Advances in Modeling Earth Systems*, 13(7):e2021MS002477, 2021.

[70] Junqi Yin, Feiyi Wang, and Mallikarjun Shankar. Strategies for integrating deep learning surrogate models with hpc simulation applications. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 01–10. IEEE, 2022.

[71] Carlos HS Barbosa, Liliane NO Kunstmann, Rômulo M Silva, Charlan DS Alves, Bruno S Silva, MS Djalma Filho, Marta Mattoso, Fernando A Rochinha, and Alvaro LGA Coutinho. A workflow for seis-

mic imaging with quantified uncertainty. *Computers & Geosciences*, 145:104615, 2020.

[72] Franz Poeschel, Juncheng E, William F Godoy, Norbert Podhorszki, Scott Klasky, Greg Eisenhauer, Philip E Davis, Lipeng Wan, Ana Gainaru, Junmin Gu, et al. Transitioning from file-based hpc workflows to streaming data pipelines with openpmd and adios2. In *Smoky Mountains Computational Sciences and Engineering Conference*, pages 99–118. Springer, 2021.

[73] Rubén Langarita Benítez. Evaluation of genome alignment workflows on hpc processors. Master's thesis, Universitat Politècnica de Catalunya, 2021.

[74] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[75] Michael Laufer and Erick Fredj. High performance parallel i/o and in-situ analysis in the wrf model with adios2. *arXiv preprint arXiv:2201.08228*, 2022.

[76] Christian Witzler, J Miguel Zavala-Aké, Karol Sierociński, and Herbert Owen. Including in situ visualization and analysis in pdi. In *International Conference on High Performance Computing*, pages 508–512. Springer, 2021.

[77] Imdb dataset. https://www.imdb.com/interfaces/.

[78] Shaker H Ali El-Sappagh, Abdeltawab M Ahmed Hendawi, and Ali Hamed El Bastawissy. A proposed model for data warehouse etl processes. *Journal of King Saud University-Computer and Information Sciences*, 23(2):91–104, 2011.

[79] *US Census Data*, August 2022.

[80] Jean-loup Gailly and Mark Adler. Gnu gzip. *GNU Operating System*, 1992.

[81] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. Igraph: An incremental data processing system for dynamic graph. 10(3):462–476, jun 2016.

[82] Rafael Vescovi, Hanyu Li, Jeffery Kinnison, Murat Keçeli, Misha Salim, Narayanan Kasthuri, Thomas D Uram, and Nicola Ferrier. Toward an automated hpc pipeline for processing large scale electron microscopy data. In *2020 IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*, pages 16–22. IEEE, 2020.

[83] Michael Salim, Thomas Uram, J Taylor Childers, Venkatram Vishwanath, and Michael Papka. Balsam: Near real-time experimental data analysis on supercomputers. In *2019 IEEE/ACM 1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing (XLOOP)*, pages 26–31. IEEE, 2019.

[84] Stephen Hudson, Jeffrey Larson, John-Luke Navarro, and Stefan M Wild. libensemble: A library to coordinate the concurrent evaluation of dynamic ensembles of calculations. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):977–988, 2021.

[85] Alexander Brace, Michael Salim, Vishal Subbiah, Heng Ma, Murali Emani, Anda Trifa, Austin R Clyde, Corey Adams, Thomas Uram, Hyunseung Yoo, et al. Stream-ai-md: Streaming ai-driven adaptive molecular simulations for heterogeneous computing platforms. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–13, 2021.

[86] Narayan Desai. Cobalt: an open source platform for hpc system software research. In *Edinburgh BG/L System Software Workshop*, pages 803–820, 2005.

[87] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. Cxl memory as persistent memory for disaggregated hpc: A practical approach. *arXiv preprint arXiv:2308.10714*, 2023.

[88] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost modelling for optimal data placement in heterogeneous main memory. *Proceedings of the VLDB Endowment*, 15(11):2867–2880, 2022.

[89] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. Failure tolerant training with persistent memory disaggregation over cxl. *IEEE Micro*, 43(2):66–75, 2023.

[90] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. Halo: A hybrid pmem-dram persistent hash index with fast recovery. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1049–1063, 2022.

[91] Jan Kończak and Paweł T Wojciechowski. Failure recovery from persistent memory in paxos-based state machine replication. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 88–98. IEEE, 2021.

[92] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. Memory pooling with cxl. *IEEE Micro*, 43(2):48–57, 2023.

[93] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. Performance evaluation on cxl-enabled hybrid memory pool. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–5. IEEE, 2022.

[94] KyungSoo Lee, Sohyun Kim, Joohee Lee, Donguk Moon, Rakie Kim, Honggyu Kim, Hyeongtak Ji, Yunjeong Mun, and Youngpyo Joo. Improving key-value cache performance with heterogeneous memory tiering: A case study of cxl-based memory expansion. *IEEE Micro*, pages 1–11, 2024.

[95] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *USENIX Annual Technical Conference*, pages 715–728, 2021.

[96] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019.

[97] Sai Sha, Chuandong Li, Xiaolin Wang, Zhenlin Wang, and Yingwei Luo. Hardware-software collaborative tiered-memory management framework for virtualization. *ACM Trans. Comput. Syst.*, 42(1–2), feb 2024.

[98] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash 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*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[99] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon 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*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[100] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh 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*, ASPLOS '20, page 875–890, New York, NY, USA, 2020. Association for Computing Machinery.

[101] Jie Ren, Dong Xu, Ivy Peng, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Hm-keeper: Scalable page management for multi-tiered large memory systems. *arXiv preprint arXiv:2302.09468*, 2023.

[102] Sandeep Kumar, Aravinda Prasad, Smruti R Sarangi, and Sreenivas Subramoney. Radiant: efficient page table management for tiered memory systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 66–79, 2021.

[103] Tong Jin, Fan Zhang, Qian Sun, Hoang Bui, Melissa Romanus, Norbert Podhorszki, Scott Klasky, Hemanth Kolla, Jacqueline Chen, Robert Hager, et al. Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1033–1042. IEEE, 2015.

[104] Harald Servat, Antonio J Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing*, pages 126–136. IEEE, 2017.

[105] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2):20–29, 2023.

[106] Jie Ren, Dong Xu, Ivy Peng, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Rethinking memory profiling and migration for multi-tiered large memory systems, 2023.