A Quantitative Approach for Adopting Disaggregated Memory in HPC Systems

Jacob Wahlgren
jacobwah@kth.se
KTH Royal Institute of Technology
Sweden

Gabin Schieffer
gabins@kth.se
KTH Royal Institute of Technology
Sweden

Maya Gokhale
gokhale2@llnl.gov
Lawrence Livermore National Laboratory
USA

Ivy Peng
ipeng@acm.org
KTH Royal Institute of Technology
Sweden

ABSTRACT

Memory disaggregation has recently been adopted in data centers to improve resource utilization, motivated by cost and sustainability. Recent studies on large-scale HPC facilities have also highlighted memory underutilization. A promising and non-disruptive option for memory disaggregation is rack-scale memory pooling, where node-local memory is supplemented by shared memory pools. This work outlines the prospects and requirements for adoption and clarifies several misconceptions. We propose a quantitative method for dissecting application requirements on the memory system from the top down in three levels, moving from general, to multi-tier memory systems, and then to memory pooling. We provide a multi-level profiling tool and *LBench* to facilitate the quantitative approach. We evaluate a set of representative HPC workloads on an emulated platform. Our results show that prefetching activities can significantly influence memory traffic profiles. Interference in memory pooling has varied impacts on applications, depending on their access ratios to memory tiers and arithmetic intensities. Finally, in two case studies, we show the benefits of our findings at the application and system levels, achieving 50% reduction in remote access and 13% speedup in BFS, and reducing performance variation of co-located workloads in interference-aware job scheduling.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**; **Emerging interfaces**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

KEYWORDS

HPC system, disaggregated memory, multi-tier memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC '23, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607108

ACM Reference Format:
Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. 2023. A Quantitative Approach for Adopting Disaggregated Memory in HPC Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607108

1 INTRODUCTION

The memory subsystem is a major consideration for cost and performance of large-scale clusters. In the last decade, the memory bandwidth wall has emerged as the bandwidth per core continues to decrease. To satisfy the increasing demand, memory capacity and bandwidth per node have increased dramatically in the last 15 years, as illustrated in Figure 1. New memory technologies like high-bandwidth memory (HBM) can provide significant bandwidth. However, the cost per bit is three to five times that of regular DDR due to a complex production process and high market demand [13]. Hence, a hybrid of DDR and HBM has become the de-facto memory configuration on today's HPC systems as we show in Table 1. For instance, the No. 1 supercomputer, Frontier, features 512 GB DDR4 and 512 GB HBM per compute node.

The node architecture in clusters and data centers today is monolithic – compute and memory resources are tightly coupled within a node's boundary. In a cloud environment, multiple jobs may share one server through virtualization, and still, recent works find that substantial node memory is not utilized [24]. In an HPC environment, the resource allocation is even more coarse-grained because jobs typically do not share a node. Combined with high memory capacity per node, this means that memory is often an under-utilized resource. Recent studies on leadership facilities have found that fewer than 15% jobs use more than 75% of the node memory, and that 50% of the time less than 12% of the total memory is in use [8, 33, 36, 37].

Recently, major cloud operators have started exploring memory disaggregation to address the challenge of memory utilization, including Meta [30] and Microsoft Azure [24]. At a high level, the idea is to have a portion of memory resources provided on-demand through remote memory pooling. In memory pool-based systems, each node has a fixed local memory and a variable amount of fabric-attached remote memory, effectively a form of a multi-tier memory system [14]. Previous works have explored using node-local persistent memory, like Intel Optane DC PM, as a second memory tier [12, 38]. Since Optane is discontinued, and with the recent

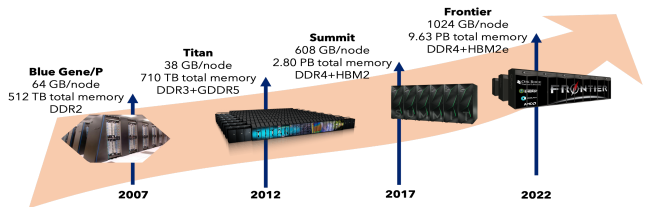


Figure 1: The evolution of memory characteristics of top leadership supercomputers in the past 15 years.

Rank	DDR/node	HBM/node	HBM BW/node	Nodes	Est. DDR cost	Est. HBM cost
Frontier [22]	512 GB	512 GB	12.8 TB/s	9,408	$ 34 M	$ 135 M
Fugaku [17]	–	32 GB	1.0 TB/s	158,976	–	$ 142 M
LUMI-G [28]	512 GB	512 GB	12.8 TB/s	2,560	$ 9.2 M	$ 35 M
Leonardo [4]	512 GB	256 GB	8.2 TB/s	3,456	$ 12 M	$ 25 M
Summit [47]	512 GB	96 GB	5.4 TB/s	4,608	$ 17 M	$ 12 M
Sierra [21]	256 GB	64 GB	3.6 TB/s	4,284	$ 7.7 M	$ 7.7 M
Sunway [9]	32 GB	–	–	40,960	$ 9.2 M	–
Perlmutter (GPU) [34]	256 GB	160 GB	6.2 TB/s	1,536	$ 2.8 M	$ 7.0 M
Selene [35]	1 TB	640 GB	16 TB/s	280	$ 2 M	$ 4.9 M
Tianhe-2A [10]	192 GB	–	–	16,000	$ 21.6 M	–

Table 1: A summary of memory configuration on Top 10 supercomputers and the estimated memory cost based on HBM having 3×-5× unit price of DDR [13].

advances in cache-coherent interconnect protocols, i.e., Compute Express Link (CXL) type 3 devices, memory pooling becomes an emerging option for implementing the second tier [44].

Existing works in multi-tier systems with memory pooling focus on cloud infrastructures [18, 19, 24, 26, 27, 30, 39]. Cloud providers are motivated to guarantee a certain Quality of Service (QoS) while minimizing their cost, e.g. through increased system utilization. While their findings and solutions are useful for informing HPC systems, HPC-oriented studies are needed for factoring in differences in the deployment environment, user expertise, and expectation. Therefore, this work aims to lay out the general and practical considerations for adopting disaggregated memory on HPC systems.

Although disaggregated memory is a type of multi-tier memory, it differs from traditional Non-Uniform Memory Access (NUMA) systems, which are symmetric, with each NUMA domain containing the same compute and memory resources, and optimizations focus on moving computing to cores in the same domain as the accessed memory pages. Instead, a tiered memory system could be asymmetric, where some tiers are CPU-less, and optimizations focus on placing hot pages in faster tiers [24].

We propose a three-level quantitative approach for dissecting application requirements on the memory system from the top down. At the top level, an application's intrinsic requirements on the memory subsystem are captured and these properties remain preserved across different memory systems. The second level quantifies the impact of a general multi-tier memory system and captures the memory access ratios across different tiers. Finally, the third level quantifies the impact of memory interference, a specific challenge of pooling-based multi-tier memory.

Several existing works focus on understanding the performance degradation or optimizing the data placement on disaggregated memory [18, 24, 30, 39, 48]. However, instead of providing yet another optimization work of data placement, this paper builds a generic framework to study an application's memory behavior and identify optimization priority and limits, as well as influence deployment options. To facilitate the quantitative study, we provide a multi-level profiling tool and also *LBench* , a benchmark for quantifying the impact of memory interference. Our methodology is applied in a set of diverse applications, including NekRS, SuperLU, Hypre, HPL, BFS, and XSBench. Finally, in two case studies, we show that findings from our approach can guide application-level optimizations on data placement, reducing remote access by 50% and improving the performance of BFS by 13%, as well as guide system-level scheduling to reduce performance variation of co-located jobs.

We summarize our contributions in this work as follows:

- We describe the prospects and requirements for adopting disaggregated memory in HPC systems.
- We propose a quantitative method for dissecting application requirements on memory systems in three levels, from general to multi-tier memory systems and memory pooling.
- We develop a multi-level profiler for facilitating the quantitative method and *LBench* for quantifying interference on memory pooling.
- We evaluate our method in NekRS, SuperLU, Hypre, HPL, BFS, and XSBench on multiple emulated system configurations and identified key insights.
- We demonstrate the usage of our method in two case studies, for optimizing the memory access ratios at the application level and interference-aware job scheduling at the system level.

2 MOTIVATION AND BACKGROUND

In the past 15 years, as workloads on HPC systems continue evolving, high computation throughput, massive datasets, complex workflow, and emerging machine learning components drive up the requirements on the memory system (see Figure 1). Table 1 summarizes the memory configuration on today's top 10 supercomputers [42], where memory is becoming a major cost factor. Meanwhile, studies on multiple HPC facilities, including NERSC's Cori supercomputer [33] and Livermore Computing's clusters [37], have shown that the utilization of memory resources can be as low as only 15% of scientific workloads utilizing at least 75% memory resources. In data centers, under-utilization of memory has motivated major cloud providers to leverage disaggregated memory to provide memory resources as needed and improve overall utilization [19, 24, 30].

Disaggregated Memory refers to a type of architecture that decouples memory resources from compute resources so that a flexible amount of memory resources can be provisioned to each workload. In contrast, all the supercomputers in Table 1 have a fixed amount of memory in DDR and HBM, so a job cannot use more than the specified memory capacity on a node, or it will face out-of-memory (OOM) errors and abort.

There are two categories of disaggregated memory architectures – split and pool [14]. In a split architecture, nodes can borrow memory from each other in a peer-to-peer fashion. In contrast, a *Memory Pool* provides a dedicated memory resource shared by multiple nodes. Memory disaggregation enables peak-of-sums provisioning rather than sum-of-peaks provisioning, reducing the required

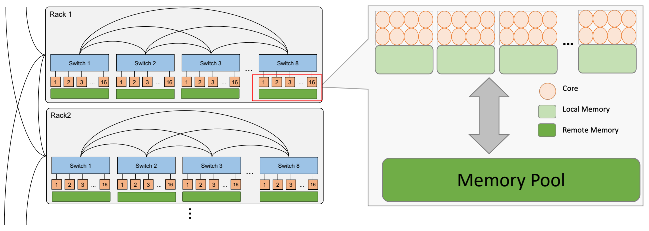


Figure 2: An HPC system architecture with rack-scale memory disaggregation. Each node has a fixed node-local memory. Nodes in the same rack also share a memory pool.

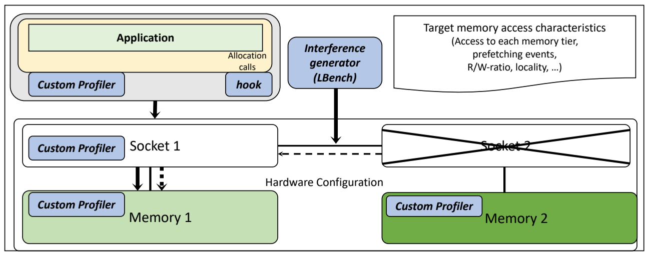


Figure 3: The overall architecture of the three-level memory-centric profiler and the emulation platform.

resources on a system level [26]. Hence, the total ownership cost (TCO) can be reduced, resulting in a great incentive for major cloud providers. Previous network-based disaggregation solutions face challenges of performance degradation. The recent development of high-performance cache-coherent interconnects, like the CXL standard, dramatically improves the feasibility and is endorsed by all major vendors. For HPC systems, a rack-scale memory pooling architecture, as illustrated in Figure 2, may be the most feasible and user-transparent design in the near term [14, 32]. This design bears similarity with the LLNL's HPE Rabbit storage design [20], where compute blades in a rack share a pool of SSD storage resources.

2.1 Challenges and Misconceptions

A disaggregated memory system is a specific implementation of *Muti-tier Memory* systems. Muti-tier memory is composed of different tiers with distinctive performance and capacity. For instance, 8 out of the top 10 supercomputers in Table 1 use HBM-DDR-based multi-tier memory systems. For the architecture presented in Figure 2, the node-local memory (light green) forms a top tier, and the memory pool (dark green) forms the bottom tier of a node's memory system.

Memory Interference in a memory pool refers to the influence from other nodes sharing the same pool. Through the shared memory pool, interference can cause unpredictable performance degradation of one job due to jobs on other nodes. For example, one cause for the performance variation is unrelated jobs on different nodes competing for the link bandwidth between compute node and memory pools.

Misconceptions of multi-tier memory include that the memory bandwidth is lower than homogeneous memory. In fact, from the hardware perspective, adding additional memory tiers (channels) can increase the aggregate bandwidth. However, a challenge is to utilize it effectively. Another misconception is that application performance will always be degraded. Distributed-memory HPC applications have the option to minimize exposure to the pooled memory tier by scaling to more compute nodes instead.

2.2 Practical Considerations for Adoption

The differences between HPC and cloud systems have to be considered for adoption. HPC applications consist of many optimized numerical kernels, run in multiple bare-metal nodes tightly coupled by MPI communication, without virtualization or node sharing. Cloud workloads like web services, and databases, are optimized for tail latency of requests. Cloud environments are virtualized and users share physical nodes. Also, the incentive structures are different. While cloud providers are motivated to reduce costs while still meeting the same service quality, HPC users are responsible for delivering the performance of their applications. Thus, approaches derived for cloud may be inapplicable on HPC, e.g., prefetching is found to be harmful in [29], but we find it necessary for HPC. Cloud solutions are mostly based on extended hypervisors or managed runtimes, while HPC infrastructures do not have such layers. Here, we highlight several practical requirements for adopting disaggregated memory in HPC systems.

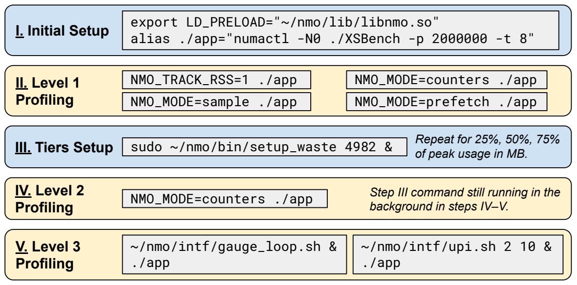
Low Porting Efforts. For wide adoption in HPC, the changes in architecture should aim to require low porting efforts. One solution for transparent porting is extending current NUMA system support. As an example, a recent kernel patch implements non-uniform interleaving policies [50] which enables applications to transparently utilize the aggregated bandwidth of tiered memory [41]. Alternatively, existing memory allocators can be extended to support multi-tier systems.

Low Performance Variation. Many runtime solutions for optimizing data placement on multi-tier memory systems are proposed. While automatic NUMA support requires low porting efforts, runtimes take time to collect enough information for decisions and are often slow in adapting to changes in access patterns, resulting in performance variation from run to run. HPC applications, however, demand more deterministic and reproducible performance.

Incentive for Computing Facility. End users of HPC systems are usually not incentivized to improve system-level resource utilization. Therefore, the computing facility needs to be the main driver for adoption. Potential incentives for facilities to adopt memory pooling include sustainability, the feasibility of separate upgrading of system components, and reduced costs.

3 METHODOLOGY

In this work, we employ a three-level top-down approach. In the first level, we identify an application's intrinsic requirements on memory systems, independent of exact system architecture. The second level extends the application requirements onto general multi-tier memory systems and leverages a memory roofline model. The final level investigates the specific memory interference challenge when the lower tier is backed by memory pooling on an HPC


I. Initial Setup
II. Level 1 Profiling
III. Tiers Setup
Repeat for 25%, 50%, 75% of peak usage in MB.
IV. Level 2 Profiling
Step III command still running in the background in steps IV–V.
V. Level 3 Profiling

```
export LD_PRELOAD="~/nmo/lib/libnmo.so"
alias ./app="numactl -N0 ./XSBench -p 2000000 -t 8"
```

```
NMO_TRACK_RSS=1 ./app
NMO_MODE=sample ./app
NMO_MODE=counters ./app
NMO_MODE=prefetch ./app
```

```
sudo ~/nmo/bin/setup_waste 4982 &
```

```
NMO_MODE=counters ./app
```

```
~/nmo/intf/gauge_loop.sh &
./app
```

```
~/nmo/intf/upi.sh 2 10 &
./app
```

Figure 4: A step-by-step overview of the profiling workflow with example commands.

architecture illustrated in Figure 2. Figure 4 presents the experimental workflow, where each level of execution collects profiling information that can be visualized separately.

3.1 Multi-level Profiling

We develop a multi-level profiler to support the three-level memory-centric analysis methodology. The profiler uses low-overhead hardware performance counters to capture application-level metrics, such as memory accesses in different levels of the memory hierarchy. Besides program-level profiling, the profiler also offers APIs, i.e., `pf_start("tag")` and `pf_stop()`, for simple tracing support to attribute results to specific kernels. This section details our methodology based on Linux and the Intel Skylake-X architecture. Key performance events are also available on other architectures, e.g. AMD's Instruction-Based Sampling and ARM's Statistical Profiling Extension. Kernel-based page-level profiling may be used as an alternative, but would require extensive changes. Our method can be adapted for general multi-tier memory (e.g. DDR+HBM, PM+DDR), only needing to change selected performance events.

Level 1: General Characteristics. The first level of profiling aims to understand an application's requirements on the memory subsystem. These include its arithmetic intensity, memory capacity usage, bandwidth usage, and access pattern. The arithmetic intensity is measured using hardware performance counters, enabling us to place the application into a roofline model. The number of bytes loaded is measured with the `OFFCORE_RESPONSE:L3_MISS` events. The offcore events include all memory loads, including hardware prefetchers. The memory capacity usage is measured by sampling the `numa_maps` file in `procfs`. The memory access pattern is measured in two ways, first using precise event-based sampling (PEBS) to record the virtual address of demand load misses. Secondly, counters related to hardware prefetching are measured to understand if the access pattern is predictable.

Level 2: Multi-Tier Memory Access. In a multi-tier environment, we define two key metrics. The *remote capacity ratio* is the ratio of lower-tier memory to total available memory. In our setup, it can be measured from `numa_maps`. The *remote access ratio* is the ratio of memory accesses to a lower tier. In our setup, it is measured using the `LOCAL_DRAM` and `REMOTE_DRAM` offcore events. The event-based sampling of memory accesses is also extended to multiple tiers by separating cache miss events to local or remote memory.

Level 3: Memory Interference. A disaggregated memory system is a specific form of multi-tier memory system. When memory pools are used to decouple memory from compute, interference from other nodes sharing the memory pool may impact performance. In Section 3.2, we present a benchmark for measuring both the interference sensitivity and induced interference of an application. The interference sensitivity determines the performance degradation due to remote memory interference, and the induced interference determines how much interference an application causes for others. We measure the injected traffic at the system level using the UPI counters `sktXtraffic` in Intel PCM. Note that link traffic may exceed the peak data bandwidth due to protocol overheads.

3.2 Measuring Memory Interference

We developed a benchmark called *LBench* based on the methodology in [7] for injecting and quantifying interference on the link to the memory pool. The benchmark allocates an array on the memory pool and performs a simple numerical kernel similar to Empirical Roofline Toolkit [51] on the array. We define the level of interference (denoted *LoI*) as the percentage of generated link traffic compared with the peak link traffic (which is 1 flop, 12 threads on our testbed). *LoI* is configured by varying the number of floating-point operations per array element in the kernel. We present a code snippet of the kernel's inner loop below.

```
if (NFLOP % 2 == 1)
    beta = A[i] + alpha;
const int NLOOP = NFLOP/2;
#pragma GCC unroll 16
for (int k = 0; k < NLOOP; k++)
    beta = beta * A[i] + alpha;
A[i] = beta;
```

By measuring the link traffic using level 3 profiling, we determine the number of flops per element corresponding to each level of intensity (i.e., $LoI = 10, 20, \ldots$). One advantage of using *LBench* over raw performance counters (PCM) is that PCM cannot measure contention beyond the saturation point. For instance, the measured traffic saturates at the link bandwidth (e.g., 85 GB/s on our testbed), while the contention will still increase due to queueing. With *LBench* , we can distinguish between saturated and contended links. Our validation results in Section 6 confirm that using *LBench* for quantifying interference can reach higher precision than raw performance counter measurement.

We also define a metric called *interference coefficient* (denoted as *IC*) by running *LBench* with one thread with 1 flop/element for a set number of iterations and measuring the relative runtime T of *LBench* . The interference coefficient is calculated as $\text{IC} = \frac{T}{T_{\text{idle system}}}$. The interference coefficient quantifies the interference induced by an application on the system.

3.3 Emulation Platform

We configure an emulation platform for the memory pool in Figure 2. The methodology uses the socket interconnect in a dual-socket system to emulate a coherent disaggregated memory system. Similar methods are used in [24, 30, 53]. The default first-touch page allocation policy places allocations onto the local NUMA node until full, before spilling to other NUMA nodes (the remote memory).

The emulation platform uses an Intel Xeon testbed with two sockets and one NUMA node per socket. As illustrated in Figure 3,

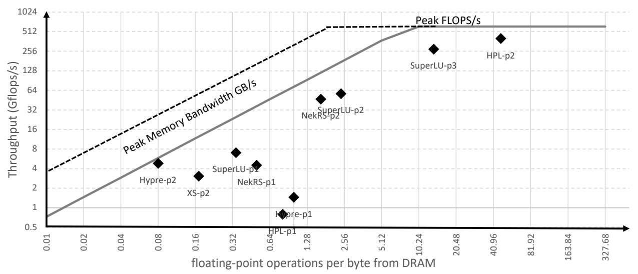


Figure 5: A roofline model built on our test platform. Dashed lines indicates extension by adding additional memory tiers.

Application	Description	Parallelization	Input Problems

HPL [11]	High Performance LINPACK benchmark, dense LU factorization with partial pivoting.	MPI+OpenMP	N=20000
			N=28280
			N=40000

Hypre [15]	Library of high-performance linear solvers. We use the structured interface.	MPI+OpenMP	EX4 10 times, n=6300, ranks=1 EX4 10 times, n=6300, ranks=2 EX4 10 times, n=6300, ranks=4
NekRS [16]	Computational fluid dynamics based on the spectral element method.	MPI	turbPipePeriodic, p=5, dt=1e-2 turbPipePeriodic, p=7, dt=6e-3 turbPipePeriodic, p=9, dt=1e-3
BFS [40]	Graph processing benchmark of the breath-first search algorithm in the Ligra framework.	OpenMP	symmetric rMat, $N=2^{24}$, $M=2^{28.24}$ symmetric rMat, $N=2^{25}$, $M=2^{29.25}$ symmetric rMat, $N=2^{26}$, $M=2^{30.25}$
SuperLU [25]	Sparse LU factorization.	MPI+OpenMP	SiO [6] (nnz=1.3M) H2O [6] (nnz=2.2M) Si34H36 [6] (nnz=5.2M)
XSBench [43]	Monte Carlo neutron transport proxy application.	MPI+OpenMP	LARGE, 2M particles, 11303 gridpoints LARGE, 2M particles, 22606 gridpoints LARGE, 2M particles, 45212 gridpoints

Table 2: Evaluated workloads with three input problems of approximately 1:2:4 memory usage ratio.

one socket represents a compute node, and the memory on the other socket represents a memory pool. The cores on the second socket are not used. The UPI interconnect between them represents the remote link. The intra-socket bandwidth is 73 GB/s and latency is 111 ns, while the inter-socket bandwidth is 34 GB/s and latency is 202 ns. The Linux kernel version is 5.14. For consistent results, we disable NUMA balancing and transparent huge pages (THP).

3.4 Analytical Models

We use the standard roofline model and an extended memory roofline model to identify performance bottlenecks and hardware limits. The roofline model [51] was proposed to model the attainable peak performance P of a program with arithmetic intensity I, defined as the number of floating-point operations per byte transferred from main memory. A computing platform is described by its peak computing power F in flop/s and peak memory bandwidth B in B/s, where $P = \min(F, B \cdot I)$. The solid line in Figure 5 presents a standard roofline model derived from the architecture peak metrics (e.g., the peak flops from clock speed and AVX-512 vectors) and STREAM benchmark results. A roofline model can be further extended to model other limiting factors. For example, the slope of memory bandwidth can be adapted to include the impact of memory interference. The dashed line in Figure 5 indicates an increase in total memory bandwidth if an additional memory tier is added to the baseline system.

For fine-grained measurement, we set our profiler to quantify the throughput in flop/s, and memory access in bytes every second. Figure 5 reports the measured arithmetic intensity and obtained throughput for each phase in the evaluated applications in Table 2. The visualization of the measurement on the roofline model shows good coverage in arithmetic intensity and throughput. Therefore, we confirm that the evaluated workloads represent HPC workloads in the memory-bound to compute-bound spectrum. Furthermore, fine-grained measurement allows us to separate distinctive phases with different arithmetic intensities. As reported in Figure 5, each application typically consists of at least two phases, where the first phase (denoted as p1) represents the initialization phase. This experiment only uses the node-local memory and no other co-running applications.

4 WORKLOAD CHARACTERIZATION

In this section, we focus on the fundamental memory requirements of an application. We are interested in identifying and abstracting the properties that persist even when the application is executed on systems with different memory configurations. Table 2 summarizes the list of evaluated applications and input problems used in this paper. Unless otherwise noted, experiments use the first listed input problem of each application.

4.1 Memory Capacity and Bandwidth Scaling

Capacity and bandwidth are the two primary memory-related considerations when deploying an HPC application. In a typical decision flow, a user needs to estimate the total memory footprint of the job and peak memory usage per node, then compare them with memory capacity per compute node to determine the minimum number of nodes required. When memory bandwidth is a limiting factor, a user may decide to increase the number of nodes further for higher aggregate memory bandwidth. This may be guided by the roofline model. Other dimensions of this decision include increased communication and core-hour cost with more nodes.

We propose a memory bandwidth-capacity scaling curve, as demonstrated in Figure 6, to help users quantify the relationship between capacity and bandwidth usage. The curve is built using the memory access sampling in our profiler. After measuring and aggregating the number of memory accesses by page, we sort pages into descending order of accesses. Then, we build the cumulative distribution of accesses to compare with the percentage of memory footprint. Each application is tested with three input problems of approximately doubling size.

The bandwidth-capacity scaling curve reveals that HPL and Hypre exhibit relatively uniform memory access across the memory footprint. Such characteristic is consistent with traditional numerical codes, where nearly all main memory objects are accessed for computation. In contrast, BFS and XSBench have only a small portion of the memory footprint being actively accessed during the execution. We check the source code and find that BFS allocates large graph structures, while only adjacency data will be accessed during execution. Similarly, XSBench allocates large grid structures while only sampled points will be looked up.

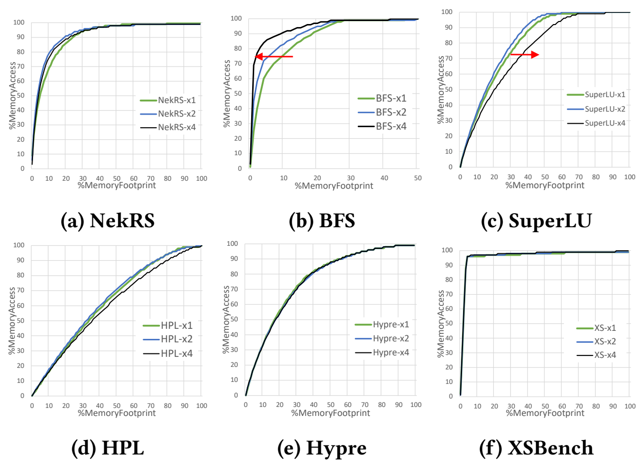


Figure 6: The cumulative distribution function of memory accesses and memory footprint in six applications at three scaled input problems.

A more interesting finding is that all applications but SuperLU exhibit a consistent bandwidth-capacity scaling pattern across increased input problems. Four applications, NekRS, HPL, Hypre, and XSBench, have overlapping scaling curves for 1×, 2×, and 4× input problems, indicating their usage patterns in bandwidth and capacity preserve across input sizes. BFS has the scaling curve shifted towards the left side of the x-axis when the input problem increases, indicating the distribution of access is becoming more skewed, with fewer percentage of pages having more accesses. The change in SuperLU indicates the distribution of hot pages moving from a skewed distribution towards a more uniform distribution. Overall, this scaling curve analysis is a concise visualization tool for HPC users to unify the two dimensions of memory requirements, i.e., bandwidth and capacity, and estimate requirements of larger-scale problems.

4.2 Hardware Prefetching

We consider the suitability of prefetching as a property determined by an application's algorithm and access pattern. Therefore, even if an application is executed on different systems, the suitability of prefetching is preserved. With disaggregated memory, prefetching may become even more important to hide the higher access latency. A benefit of non-faulting implementations, such as CXL, is that hardware prefetching can remain fully operational for remote memory.

To quantify the suitability of prefetching, we propose using two previously proposed metrics, namely *Accuracy* and *Coverage* [29]. *Accuracy* measures the ratio of prefetched cachelines that have been actually accessed by the program. *Coverage* measures the ratio of cacheline accesses that were prefetched before on-demand access. Our analysis in this work is based on L2 cache, where the core hardware prefetcher is located and LLC is an exclusive L3.

On our testbed, we set our profiler to capture measurements from four hardware counters: PF_L2_DATA_RD, PF_L2_RFO, L2_LINES_IN,

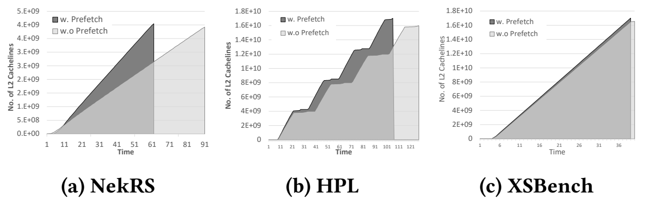


Figure 7: Measured memory traffic in number of cachelines (y-axis) and runtime (x-axis) in three applications with and without L2 prefetching enabled.

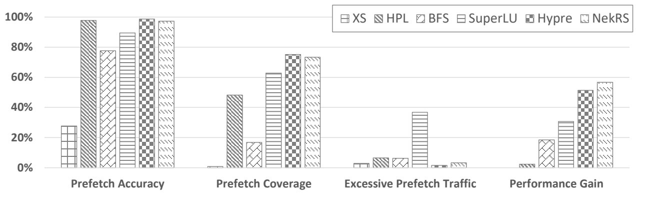


Figure 8: Accuracy, coverage, excessive memory traffic, and performance gain from prefetching in all tested applications

and USELESS_HWPF. We calculate the two metrics as follows.

$$\text{Accuracy} = \frac{\text{PF\_L2\_DATA\_RD} + \text{PF\_L2\_RFO} - \text{USELESS\_HWPF}}{\text{PF\_L2\_DATA\_RD} + \text{PF\_L2\_RFO}} \quad (1)$$

$$\text{Coverage} = \frac{\text{PF\_L2\_DATA\_RD} + \text{PF\_L2\_RFO} - \text{USELESS\_HWPF}}{\text{L2\_LINES\_IN} - \text{USELESS\_HWPF}} \quad (2)$$

Additionally, we execute the workload with prefetching disabled to determine the performance gain prefetching contributes. Hardware prefetching is disabled by configuring a model-specific register in the processor core (the two least significant bits of MSR 0x1a4).

Figure 8 reports the measured prefetching accuracy and coverage of each application. All except XSbench and BFS have more than 80% prefetching accuracy. Hypre and NekRS have the highest prefetching coverage – about 70% of L2 cacheline accesses are prefetched instead of fetched on demand. Figure 8 presents a timeline of the number of fetched cachelines with and without L2 prefetching in NekRS, HPL, and XSBench, respectively. The results show that prefetching may contribute to a substantial portion of memory bandwidth in the tested applications. For instance, Figure 7a shows that the memory bandwidth consumption in NekRS is significantly higher when prefetching is enabled. Note that the total memory traffic in NekRS with prefetching is only 3% higher than that without prefetching, but the performance gain is as high as 57%, as reported in Figure 8. A similar observation of prefetching contributing substantial memory traffic is also identified in cloud workloads [29]. However, these workloads exhibit low prefetching accuracy and coverage, and thus, prefetching is considered harmful. In contrast, the high coverage and accuracy, and performance gain in our experiments indicate the suitability of prefetching for scientific workloads.

All applications have low excessive memory traffic (2%-6% as reported in Figure 8) due to prefetching, except SuperLU where the total memory traffic with prefetching enabled is 37% higher than that with prefetching disabled. However, the performance gain is almost 31% and thus prefetching is still useful from a performance perspective. Although XSBench has the lowest accuracy, it also has low excessive memory traffic (3%) from prefetching, indicating the prefetching is automatically adapted to a low level when accuracy is low.

The memory bandwidth-capacity scaling curve unifies two main factors and aids users in projecting memory usage on different problems. Prefetching may constitute major memory usage and is critical for the performance of HPC applications.

5 MULTI-TIER MEMORY

In this section, we cover the application impact when introducing an additional memory tier. A general multi-tier memory system has relatively faster but smaller top tiers and slower but larger bottom tiers. The exact memory technologies for implementing each tier may vary. For the experiments in this section, we emulate a two-tier memory system and run the profiler on selected applications to quantify memory access to each tier.

To capture the impact of multi-tier memory on performance bottlenecks, a memory roofline model was proposed in previous work [8] to model the memory access performance as a function of the local to remote memory access (L:R) ratio. The model guides tuning towards high L:R ratios to shift the limit of the memory access performance toward the peak bandwidth of the fast-tier memory. In fact, the peak memory performance can be further increased by leveraging all memory tiers concurrently instead of solely the fast tier. Therefore, we emphasize balanced local to remote accesses that match the bandwidth and capacity limit of each memory tier.

5.1 Tiered Memory Access

Two reference points are critical for guiding optimization of applications on multi-tier memory systems. The first one is the ratio of the memory capacity of each tier. The second reference point is the ratio of memory bandwidth of each tier. First, using our profiler, we quantify the ratio of memory accesses to each memory tier. Then, we calculate R^i_{Cap}, R^i_{BW}, and R^i_{access}, respectively, to quantify the ratio of memory capacity, bandwidth, and access to a tier i and compare them with the two reference points.

Figure 9 reports the measured R^i_{access} of each phase in the tested applications on three system configurations, where $R^i_{Cap} = 25\%, 50\%, 75\%$. As a validation of our profiler, we also measured the arithmetic intensity of each phase identified in Section 3.4 by using $AI = \frac{FLOPS}{Byte_{LM}+Byte_{RM}}$, where $Byte_{LM}$ denotes the number of bytes retrieved from local memory tier and $Byte_{RM}$ denotes the number of bytes retrieved from remote memory tier. The measured arithmetic intensities are consistent with those reported in the experiment on the single-tier system in Figure 5 and are omitted due to the space limit.

Figure 9 reports the percentage of accessed bytes from the remote memory tier among all memory accesses, i.e., R^{remote}_{access}. We also add two reference lines of R^{remote}_{BW} and R^{remote}_{Cap}. R^{remote}_{BW} is the turning point of memory access bottleneck, where a R^{remote}_{access} value lower than R^{remote}_{BW} indicates that the memory access bottleneck is bound by the bandwidth of the fast tier. A R^{remote}_{access} value higher than R^{remote}_{BW}, however, indicates too many memory accesses to the slower tier so that the slower tier becomes the limit of memory access performance. Thus, the R_{BW} reference is an upper bound for tuning. The lower bound of tuning uses the reference of memory capacity ratios R_{Cap}. The ratio of memory accesses to each tier should at least match the ratio of their capacity.

By leveraging the two reference points and the memory access metrics from our profiler, optimization opportunities on multi-tier memory can be identified and prioritized. In Figure 9a, the two reference lines are close, and most applications (except HPL and XSBench) have their ratios of remote memory access close to the two optimization reference lines, indicating little optimization space, and users should not spend efforts in optimizing data placement. In contrast, Figure 9c presents plenty of optimization opportunities from the two reference lines. For instance, based on the capacity ratio, HPL, NekRS, and BFS all exhibit excessive accesses above the R_{Cap} (denoted in the dashed red line). The second phase (p2) of almost all applications is substantially above the bandwidth ratio (R_{BW}) reference. Although this indicates a large space for optimization, it could also mean an ill-balanced design of the memory tiers. Note that although all phases are included, not all carry the same weight on the total execution time.

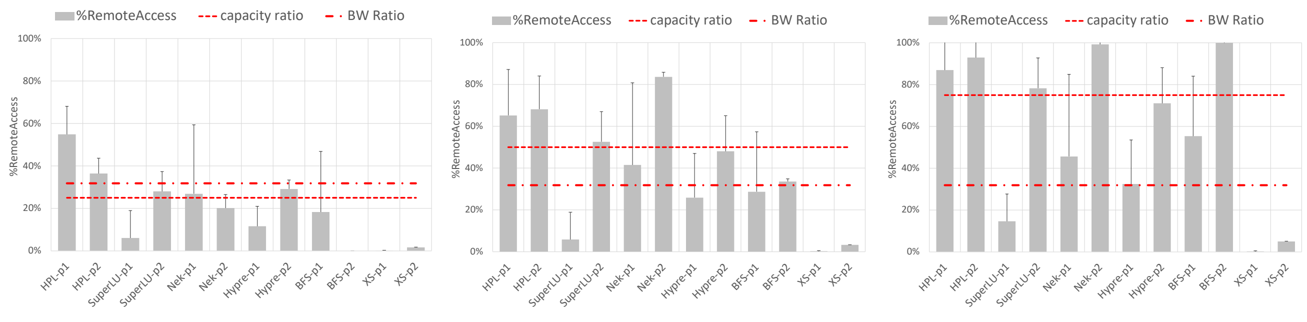
XSBench stands out among all applications in that the remote access ratio is low (below 6%) in all configurations. This means that the additional bandwidth from remote memory is not utilized. However, considering that the prefetching coverage for XSBench is $< 1\%$, the application is highly sensitive to increased memory latency. Therefore, reducing latency by minimizing remote memory exposure is more important than increasing the aggregate memory bandwidth.

5.2 Phase Changes in Memory Access

HPC applications often consist of several phases. As illustrated in Figure 5 and Figure 9, phases in an application can have very different profiles of arithmetic intensity and memory access patterns. This characteristic increases the complexity of optimizing applications on multi-tier memory systems. For instance, one scheme for a kernel's data placement may harm other kernels. This global optimization is a Knapsack problem which is NP-complete.

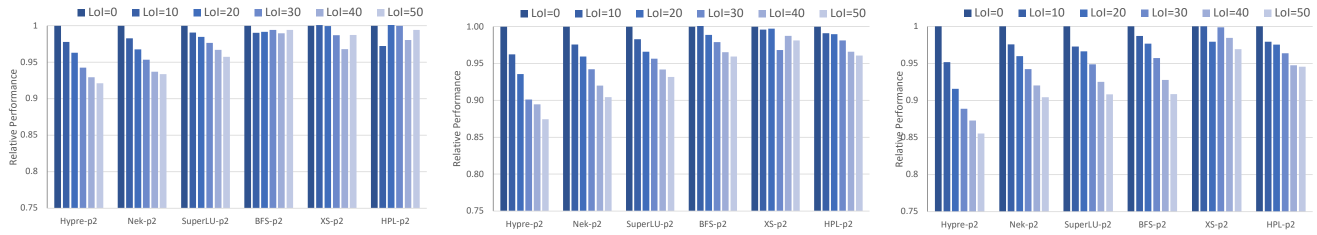
Static solutions leverage offline profiling to modify allocation sites to place a memory variable directly on a suitable memory tier. Dynamic solutions resort to runtime detection of access patterns to migrate performance-critical memory pages into the fast tier. The first direction requires extensive porting efforts and is often not adaptable to new input problems or system architecture. The second direction is user transparent. However, performance variation and indeterministic performance are often unacceptable to HPC end users. Also, on high-end HPC hardware, phases in an application are often short in time, while runtime solutions need time to adapt.

Capacity ratio and bandwidth ratio provides two optimization references and the ratio of memory access to tiers should match them. The dominant phase with unmatched memory access distribution should be the priority of optimization.



(a) 75%-25% capacity ratio **(b) 50%-50% capacity ratio** **(c) 25%-75% capacity ratio**

Figure 9: The ratio of memory accesses to the second tier on three two-tier memory systems with capacity ratios ranging from 25% to 75%. Label X-pY denotes the Y-th phase of workload X.



(a) 25%-75% capacity ratio **(b) 50%-50% capacity ratio** **(c) 25%-75% capacity ratio**

Figure 10: Quantifying application's sensitivity to memory interference on three disaggregated memory systems based on memory pool. Y-axis indicates the relative performance w.r.t LoI=0.

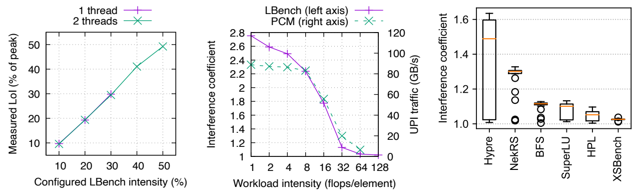


Figure 11: The scaling of interference using our benchmark (left), the comparison of measurement from our benchmark and raw performance hardware counter (middle), the interference caused by different applications (right).

6 INTERFERENCE ON MEMORY POOLING

In this section, we focus on memory pool-based disaggregated memory systems. As introduced in Section 2, disaggregated memory is a specific form of multi-tier memory. In the example in Figure 2, the bottom memory tier is implemented by a memory pool, and the top memory tier is implemented by each node's local memory. As multiple nodes may share the memory pool, a unique challenge is memory interference from co-running jobs on other nodes. We use *LBench* to quantify the impact of memory interference the applications in Table 2. For each application, we quantify two aspects of memory interference – its sensitivity to memory interference and the memory interference caused by the application. The first metric is important for HPC users to estimate their application's performance on future HPC systems equipped with the pool-based memory tier. The second metric is useful for schedulers to improve co-location decisions.

The first step of our study is to validate the accuracy and precision of *LBench* . Thus, we configure the benchmark intensity to sweep from 10 to 50 and measure the percentage of link traffic compared to the peak. As reported in the left panel of Figure 11, the measured level of intensity (y-axis) is linearly proportional to the configured intensity in *LBench* . Thus, we confirm *LBench* can be used to generate the required level of interference. Next, we compare the precision of *LBench* with the low-level hardware counters (denoted as *PCM*). We set the background workload to sweep arithmetic intensity from 1 to 128 flops per element. Then we measure the resulting interference coefficient (*IC*) as the relative runtime of *LBench* and report it in the middle panel of Figure 11. Additionally, we measure the raw link traffic from PCM. The results show that at high bandwidth usage (i.e., below 8 flops/element), *LBench* can still quantify increased contention while the hardware counter measurement saturates around 85 GB/s. The improved precision is because contention can continue increasing due to queueing effects while the measured traffic saturates at the link bandwidth. Hence, we show that *LBench* can provide accurate measurements and injection of a configurable level of intensity (*LoI*).

For the study in this section, we configure *LBench* to run with two threads for interference generation as it provides up to 50% intensity. The threads are running on the local socket to inject congestion on the link to the remote memory. We use only two threads for injection to mitigate the impact of shared L3 caches with the co-running workloads. The performance impact at 50% intensity was measured to be less than 1% for all workloads.

6.1 Sensitivity to Memory Interference

We quantify an application's sensitivity to memory interference on memory pooling using *LBench* . We configure *LBench* to generate interference of increased intensity levels, i.e., $LoI = 0, 10, \ldots$. Then, for each application, we use its performance at $LoI = 0$ as the baseline and its relative performance at higher LoI as the measurement of its sensitivity to interference. A similar approach is also used in previous work [49].

Figure 10 reports the measured sensitivity of each application to interference on the memory pool on three system configurations, i.e., the ratios between local memory capacity and remote memory pool ranges from 25%, 50%, and 75%. In general, all applications show reduced performance at an increased level of memory interference. Hypre and NekRS are among the most sensitive applications to memory interference. On the 50-50% tiers setup, Hypre and NekRS show 15% and 13% performance loss at $LoI = 50$. In Figure 9, the memory access to the remote tier (i.e., memory pooling in this case) is not the highest compared to other applications. However, due to their low arithmetic intensity, as quantified in Figure 5, they show higher sensitivity than other applications. In contrast, results in Figure 9 show that HPL has high accesses to the memory pool, while having a low sensitivity to memory interference. On the 50-50% tiers setup, it shows less than 5% performance loss even at the highest level of interference.

An application's sensitivity to memory interference on memory pooling is caused by its remote memory access and is inversely influenced by its arithmetic intensity. A quantification of application sensitivity to memory interference on a target disaggregated memory system is necessary for application users to determine its deployment configurations on the system. For applications with low sensitivity to memory interference, users can configure the job deployment to leverage more capacity from the memory pool and reduce the number of compute nodes needed to support the job. For highly sensitive applications, the users can choose to deploy a job with more compute nodes to reduce the remote memory access or even completely avoid remote memory to fulfill their performance requirements. Our quantification method provides a tool for HPC end users to make informed decisions and tradeoffs, improving user confidence in the new system architecture.

6.2 Interference Coefficient

We propose a second metric, the *Interference Coefficient*, to measure the interference caused by an application on the memory pool. As an application accesses the remote memory tier, it also injects memory traffic onto the shared pool, causing interference on the system and other jobs. The main difference from the first metric, i.e., sensitivity, is that this metric is solely related to the remote memory access but is not directly influenced by the arithmetic intensity of the application. As the sensitivity of an application is important for HPC end users, this metric is important for system-level coordination. For example, the interference coefficient could be provided as a hint in job descriptions to enable schedulers with interference awareness to improve co-location decisions.

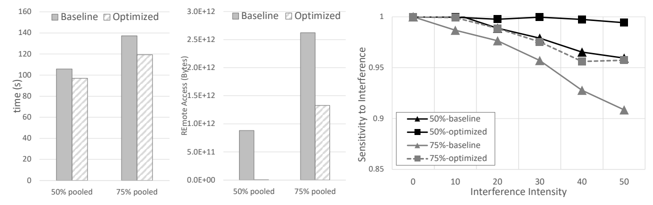


Figure 12: Optimization on data placement in BFS improves runtime, reduces remote memory accesses and sensitivity to memory interference.

We quantify the interference coefficient of applications on a target system by co-running the application with *LBench* and calculating the relative slowdown compared to an idle system. The right panel of Figure 11 reports the measured interference coefficient for the applications on a 50% memory pooling setup. The results show NekRS and Hypre can introduce the most memory interference to other co-running jobs while HPL and XSBench introduce the lowest interference. The spread in Figure 11 highlights the variance in the interference coefficient caused by a workload. For instance, in Hypre, the compute phase causes high interference coefficient, while the initialization phase causes a low interference coefficient.

An application's sensitivity to memory interference is determined by the arithmetic intensity and the amount of accesses to memory pools. Users need to incorporate the sensitivity to memory interference in deployment decisions.

7 USE CASES

In this section, we demonstrate use cases for leveraging the results of the three-level quantitative study to guide optimization at both the application level and the system level.

7.1 Optimizing Remote Memory Traffic

In the first case study, we show that analyzing the memory access distribution on memory tiers can guide programmers to prioritize the optimization of data placement at the application level. The results in the multi-tier analysis of BFS with 75% remote capacity showed 99% remote memory access – the remote access ratio is much higher than the capacity ratio reference. The mismatch indicates that the most accessed data structures are in the remote memory. Coupled with information obtained from memory allocation sites in our profiler, we identified major memory objects and found that the `Parents` array is small but highly accessed.

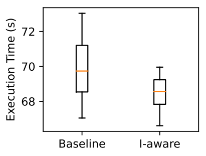
We consider three options to change its placement into the node-local memory. First, explicitly allocate in local memory, which is straightforward with libnuma. However, in the case of BFS, several large objects are allocated before `Parents`, leaving no space in local memory at the allocation site of `Parents`. Alternatively, we can explicitly allocate less accessed objects in remote memory to free up space. The second option is to migrate to local memory after initialization by exchanging pages between the local and remote memory using libnuma. However, this is not feasible to implement at the application level since it requires information on all other memory objects in local memory, such as size and access intensity. Thus, this option is often implemented at the system software level. The third option is application specific, by changing the order of allocations. By allocating and initializing objects in order of hotness, the hottest objects will be placed in local memory due to the first-touch policy. Note that this option is only suitable for applications where all memory objects have a similar lifespan (usually the whole program duration). With frequent dynamic allocations and deallocations, this option may cause the local memory to be under-utilized.

We choose to change the allocation order in BFS so that `Parents` is allocated and initialized first. With this simple change, the remote access ratio is reduced from 99% to 80% in the 75% remote memory scenario, resulting in a 6% performance improvement. Further re-applying the multi-tier memory traffic analysis shows that now many remote accesses come from dynamic heap allocations. In BFS, these are used for temporary structures, including the current frontier. Since they are dynamic in scope and size, they cannot be pre-allocated like the parents. We attempted to reserve space in the local memory by allocating a block of memory at the start and freeing it at the end of the initialization. However, this did not yield a significant benefit.

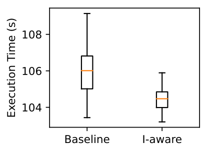
Instead, by inspecting the code, we identified a temporary object used during initialization but not afterward. The original code leaves the object unfreed due to a performance bug in the (de)-allocator. Indeed, freeing the object degrades performance by 3% when running in local memory only. However, with the memory pool as the secondary tier, freeing up the object reserves more local memory for dynamic allocations, offsetting the performance overhead of deallocation. As shown in Figure 12, this 1-line change further reduced the remote access ratio from 80% to 50%, resulting in a 13% performance improvement over the baseline at 75% memory pooling. At 50% pooling, the optimized version almost eliminates remote memory access as shown in the middle panel of Figure 12. We re-evaluate the application's sensitivity to memory interference as in Section 6 using the optimized version for the cases of 50% and 75% pooling. We compare the interference sensitivity in the right panel of Figure 12. The results show that the optimized version has a significantly reduced interference sensitivity. In the general case, the interfaces provided by Linux are insufficient for this kind of data placement optimization. An interface for reserving a portion of local memory for dynamic allocation would empower developers to optimize their applications for disaggregated memory.

7.2 Interference-Aware Job Scheduling

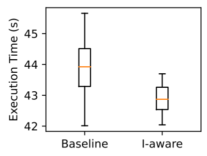
The results in Section 6.1 show that the performance impact of memory interference differs between different workloads. For practical adoption in HPC systems, job schedulers can be extended to take memory interference sensitivity into account for job co-location decisions. For instance, the user can use *LBench* and the quantification method in Section 6 to quantify the application's sensitivity and provide it at job submission. Without interference awareness, interference-sensitive workloads may be scheduled



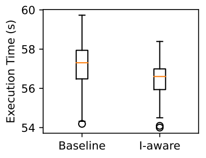
(a) NekRS



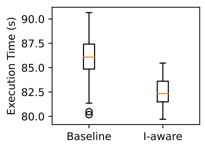
(b) BFS



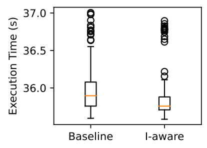
(c) SuperLU



(d) HPL



(e) Hypre



(f) XSBench

Figure 13: The five-number summary of the execution time of each application in 100 runs with a mixture of co-running applications using a random baseline and interference-aware optimization.

together using the same memory pool, leading to poor performance. Interference-aware scheduling has shown benefit in previous works targeting other types of shared resources or configurations [7, 12, 31, 46, 52, 54].

To study the impact of interference-aware scheduling, we use the emulation setup and workloads from before with 50% memory pool capacity. To simulate other jobs being scheduled onto the memory pool, we inject link traffic using *LBench* . The level of interference changes randomly between 0–50% every 60 s, representing different types of workloads. To simulate an interference-aware scheduler, which prevents interference-inducing jobs from being co-located, we vary the level of interference between 0–20% instead. Each workload is run 100 times in both configurations.

The execution time in the baseline and interference-aware scheduler for each workload is shown in Figure 13. In general, interference-aware scheduling improves execution time and reduces performance variability. The execution time is reduced because the average interference is lower with interference-aware scheduling. The variability is reduced because the range of possible interference levels is smaller with interference awareness as the high end of the spectrum is cut off. However, the results vary among the set of evaluated workloads. The average speedup is 4% for Hypre, 2% for NekRS and SuperLU, 1% for BFS and HPL, and 0% for XSBench. As a measure of variability, the 75th percentile of execution time decreased by 5% for Hypre, 3% for NekRS and SuperLU, 2% BFS, and 1% for HPL and XSBench.

As expected, these results largely match the measured interference coefficients for the workloads shown in Figure 11 and the interference sensitivity in Figure 10b. Hypre has the highest interference sensitivity and benefits most from interference-aware scheduling. Meanwhile, XSBench and HPL have the lowest interference sensitivity and show little benefit from interference-aware scheduling. The previous results showed that NekRS is more interference sensitive than BFS, but they show equal benefits from interference awareness.

These results indicate that integrating our quantitative methodology of memory interference into job schedulers such as SLURM could reduce interference and improve performance in a disaggregated memory system. The improvement of performance and reduced variation could incentivize users to inform their application's interference profile to the job scheduler and increase their confidence in the new memory architecture. Also, given that a 4% performance improvement was observed for Hypre in our emulated environment, in an environment with higher induced interference, e.g., with more than two nodes per memory pool, the performance improvement could be more significant. We recognize that a real disaggregated system may have an even more skewed latency distribution than the one emulated in our study. Such a distribution has been found to have a high impact on some types of workloads and will be better studied using more accurate simulators such as gem5 and FPGA-based emulators [2, 5]. Nevertheless, our experiments with LBench still indirectly reflect the impact of random long latency induced by multiple jobs sharing a memory pool.

8 RELATED WORKS

In this section we summarize related works on multi-tier memory systems and disaggregated memory.

Disaggregated Memory Systems. Both faulting methods (page-swapping) and non-faulting methods (direct cacheline access) have been proposed for implementing memory disaggregation [26]. Until recently, network-based page-swapping has been the main practical option [3, 19, 27]. With major vendors adopting the CXL standard, many recent works focus on realizing non-faulting CXL-based memory disaggregation with a focus on cloud systems. Pinto et al. developed a full-stack prototype based on CXL-predecessor OpenCAPI using FPGAs, and Gouk et al. developed a similar prototype for CXL [18]. The implications of CXL-based memory for cloud workloads were studied in Meta datacenters [30] and Microsoft Azure datacenters [24]. Ding et al. proposed an extended memory roofline model for designing HPC systems with disaggregated memory [8]. They used analytical methods to estimate the local to remote memory access ratio in a set of AI trainings, data analytics workloads, and HPC micro-benchmarks. The access ratio was used in a roofline model to guide system design. Wahlgren et al. proposed an emulation method for CXL-based memory and focused on evaluating the performance penalty in HPC workloads [48]. In contrast, our method is based on extensive quantitative measurements of memory access, application performance, and memory interference in HPC applications.

Memory Interference and Contention. Tudor et al. analyzed the impact of interference on parallel applications on multi-socket systems and developed an analytical model for memory contention based on the queuing theory [45]. In infrastructures with workload co-location, prior works have investigated interference from shared hardware resources, such as L3 cache, network, etc, and their impact on performance and job scheduling policies [7, 46, 54]. Recently, Masouros et al. developed an interference-aware cloud scheduler for disaggregated memory systems [31], and Lee et al. developed optimizations to mitigate interference for virtual machines with disaggregated memory [23]. Zacarias et al. developed a prediction model to estimate the performance degradation from interference in a disaggregated memory system [52, 53]. They target a split architecture, where remote memory is provided by other compute nodes rather than dedicated memory servers, different from our target architecture in this work.

Data Placement in Multi-Tier Memory. Extensive works have proposed automatic and transparent data placement solutions in multi-tier memory systems implemented with non-volatile memory or CXL. Agarwal and Wenisch proposed Thermostat, a fully application-transparent data placement runtime [1] that detects hotness of pages for migration. Duraisamy et al. evaluated a transparent runtime and optimized cluster scheduler in a production datacenter [12]. For CXL-based memory, Maruf et al. proposed TPP for transparent page placement for direct-attached memory [30], and Li et al. proposed Pond for data placement in shared memory pools [24].

9 CONCLUSIONS

The recent adoption of disaggregated memory in several major data centers and the severe memory underutilization and high cost of memory in HPC facilities have mandated a closer look into the practical adoption of rack-scale memory pooling as a non-disruptive option on future HPC systems. This work described the prospects and requirements for adoption. In particular, we presented a multi-level quantitative methodology for dissecting application requirements on memory systems, from general to multi-tier, and memory pooling. We applied our method in NekRS, SuperLU, Hypre, HPL, BFS, and XSBench and identified key insights. In two case studies, we also demonstrated how findings from our method could be used for optimizing memory access at the application level and interference-aware job scheduling at the system level.

10 ACKNOWLEDGMENTS

This research is supported by the European Commission under the Horizon project OpenCUBE (101092984) and the Swedish Research Council (no. 2022.03062). This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344 with support from the DOE Exascale Computing Project.

REFERENCES

[1] Neha Agarwal and Thomas F Wenisch. 2017. 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.* 631–644.

[2] Maryam Babaie, Ayaz Akram, and Jason Lowe-Power. 2023. Enabling Design Space Exploration of DRAM Caches for Emerging Memory Systems. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 340–342.

[3] Maciej Bielski, Christian Pinto, Daniel Raho, and Renaud Pacalet. 2016. Survey on memory and devices disaggregation solutions for HPC systems. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES).* IEEE, 197–204.

[4] Cineca. 2022. Leonardo HPC System. https://leonardo-supercomputer.cineca.eu/hpc-system/.

[5] Hüsrev Cılasun, Christopher Macaraeg, Ivy Peng, Abhik Sarkar, and Maya Gokhale. 2022. FPGA-accelerated simulation of variable latency memory systems. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '22).*

[6] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (nov 2011), 1–25. https://doi.org/10.1145/2049662.2049663

[7] Christina Delimitrou and Christos Kozyrakis. 2013. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE, 23–33.

[8] Nan Ding, Samuel Williams, Hai Ah Nam, Taylor Groves, Muaaz Gul Awan, LeAnn Lindsey, Christopher Daley, Oguz Selvitopi, Leonid Oliker, and Nicholas Wright. 2022. Methodology for Evaluating the Potential of Disaggregated Memory Systems. In *2022 IEEE/ACM International Workshop on Resource Disaggregation in High-Performance Computing (REDIS)*. 1–11.

[9] Jack Dongarra. 2016. Report on the Sunway TaihuLight System. https://netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf.

[10] Jack Dongarra. 2017. Report on the Tianhe-2A System. https://icl.utk.edu/files/publications/2017/icl-utk-970-2017.pdf.

[11] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[12] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

[13] By Kim Eun-jin. 2023. Samsung and SK Hynix Enjoy a Rush of Orders for New Memories. http://www.businesskorea.co.kr/news/articleView.html?idxno=109380. *Business Korea* (2023).

[14] Mohammad Ewais and Paul Chow. 2023. Disaggregated Memory in the Datacenter: A Survey. *IEEE Access* (2023). Publisher: IEEE.

[15] Robert D Falgout and Ulrike Meier Yang. 2002. hypre: A library of high performance preconditioners. In *Computational Science—ICCS 2002: International Conference Amsterdam, The Netherlands, April 21–24, 2002 Proceedings, Part III*. Springer, 632–641.

[16] Paul Fischer, Stefan Kerkemeier, Misun Min, Yu-Hsiang Lan, Malachi Phillips, Thilina Rathnayake, Elia Merzari, Ananias Tomboulides, Ali Karakus, Noel Chalmers, et al. 2022. NekRS, a GPU-accelerated spectral element Navier–Stokes solver. *Parallel Comput.* 114 (2022), 102982.

[17] Fujitsu. 2020. Supercomputer Fugaku – Specifications. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/.

[18] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap.. In *NSDI*. 649–667.

[20] HPCWire. 2021. Livermore's El Capitan Supercomputer to Debut HPE 'Rabbit' Near Node Local Storage. https://hpc.llnl.gov/livermoreâĂŹs-el-capitan-supercomputer-debut-hpe-rabbit-âĂŸnear-nodeâĂŹ-storage.

[21] Lawrence Livermore National Laboratory. 2018. Sierra. https://hpc.llnl.gov/hardware/compute-platforms/sierra.

[22] Oak Ridge National Laboratory. 2021. Frontier User Guide. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html.

[23] Jinhoon Lee, Yeonwoo Jung, Suyeon Lee, Safdar Jamil, Sungyong Park, Kwangwon Koh, Hongyeon Kim, Kangho Kim, and Youngjae Kim. 2023. MFence: Defending Against Memory Access Interference in a Disaggregated Cloud Memory Platform. (2023).

[24] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[25] Xiaoye S. Li. 2005. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Softw.* 31, 3 (sep 2005), 302–325. https://doi.org/10.1145/1089014.1089017

[26] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[27] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

[28] LUMI. 2022. LUMI Documentation – Hardware. https://docs.lumi-supercomputer.eu/hardware/compute/lumig/.

[29] Suyash Mahar, Hao Wang, Wei Shu, and Abhishek Dhanotia. 2023. Workload Behavior Driven Memory Subsystem Design for Hyperscale. *arXiv preprint arXiv:2303.08396* (2023).

[30] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. 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*. 742–755.

[31] Dimosthenis Masouros, Christian Pinto, Michele Gazzetti, Sotirios Xydis, and Dimitrios Soudris. 2023. Adrias: Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 855–869.

[32] Satoshi Matsuoka, Jens Domke, Mohamed Wahib, Aleksandr Drozd, and Torsten Hoefler. 2023. Myths and Legends in High-Performance Computing. *arXiv preprint arXiv:2301.02432* (2023).

[33] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh, Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf. 2022. A case for intra-rack resource disaggregation in hpc. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–26.

[34] National Energy Research Scientific Computing (NERSC). 2021. Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/.

[35] NVIDIA. 2020. NVIDIA Selene: Leadership-Class Supercomputing Infrastructure. https://www.nvidia.com/en-us/on-demand/session/supercomputing2020-sc2019/.

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

[37] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the memory underutilization: Exploring disaggregated memory on hpc systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 183–190.

[38] Ivy Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. 2020. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 916–925.

[39] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 868–880.

[40] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. *ACM SIGPLAN Notices* 48, 8 (aug 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[41] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. *arXiv preprint arXiv:2303.15375* (2023).

[42] TOP500.org. 2022. Top500 List November 2022. https://www.top500.org/lists/top500/2022/11/

[43] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[44] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 33–48.

[45] Bogdan Marius Tudor, Yong Meng Teo, and Simon See. 2011. Understanding off-chip memory contention of parallel programs in multicore systems. In *2011 International Conference on Parallel Processing*. IEEE, 602–611.

[46] Achilleas Tzenetopoulos, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. 2022. Interference-aware workload placement for improving latency distribution of converged HPC/Big Data cloud infrastructures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International Conference, SAMOS 2021, Virtual Event, July 4–8, 2021, Proceedings*. Springer, 108–123.

[47] Summit user Guide. 2018. Summit. https://docs.olcf.ornl.gov/systems/summit_user_guide.html.

[48] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing* (2022).

[49] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*.

[50] Johannes Weiner. 2022. [PATCH] mm: mempolicy: N:M interleave policy for tiered memory nodes. https://lore.kernel.org/linux-mm/YqD0%2FtzFwXvJ1gK6@cmpxchg.org/T/

[51] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. Publisher: ACM New York, NY, USA.

[52] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. 2021. Improving hpc system throughput and response time using memory disaggregation. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 283–290.

[53] Felippe Vieira Zacarias, Rajiv Nishtala, and Paul Carpenter. 2020. Contention-aware application performance prediction for disaggregated memory systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*. 49–59.

[54] Felippe Vieira Zacarias, Vinicius Petrucci, Rajiv Nishtala, Paul Carpenter, and Daniel Mossé. 2021. Intelligent colocation of HPC workloads. *J. Parallel and Distrib. Comput.* 151 (2021), 125–137.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

Memory disaggregation has recently been adopted in several major data centers to improve resource utilization, driven by cost and sustainability. Meanwhile, recent studies on large-scale HPC facilities have also highlighted memory underutilization. A promising and non-disruptive option for memory disaggregation is rack-scale memory pooling, where node-local memory is supplemented by shared memory pools. This work outlines the prospects and requirements for adoption and clarifies several misconceptions. We propose a quantitative method for dissecting application requirements on the memory system from the top down in three levels, moving from general, to multi-tier memory systems, and then to memory pooling. We also provide a multi-level profiling tool and LBench to facilitate the quantitative approach. We evaluated a set of representative HPC workloads on an emulated platform. Our results show that prefetching activities can significantly influence memory traffic profiles. Interference in memory pooling has varied impacts on applications, depending on their access ratios to memory tiers and arithmetic intensities. Finally, findings from our method are applied in two case studies to show benefits at the application level and system level, achieving 50% reduction in remote access and 13% speedup in BFS, and reducing performance variation of co-located workloads in interference-aware job scheduling.

The computational artifacts are:

- The profiling tool, with four modes: RSS (working set size), perp (Periodic Profiling of arithmetic intensity and memory access counters), samp (sampling of memory access virtual addresses), pf (Periodic Profiling of prefetching counters).
- LBench, which can be run either in continuous mode to generate background traffic, or in constant mode (fixed iteration count) to measure the interference coefficient.
- setup_waste, a tool to configure the local memory capacity in the emulated system.
- A patch to Ligra BFS improving the performance on multi-tier memory.

Additional tools used include Intel PCM and numactl. The evaluated workloads are described in the paper. By utilizing the computational artifacts, all the results in the paper can be reproduced, either to confirm the correctness or to extend the analysis to additional workloads.

REPRODUCIBILITY OF EXPERIMENTS

The workflow for carrying out the experiments is listed below.

- First run with RSS profiler. Calculate 25,50,75% of reported peak working set, to be used with setup_waste below. (10 min)
- Fig 4: Run with perp profiler. (10 min)
- Fig 5: Run with samp profiler. (10 min)
- Fig 6-7: Run with pf profiler. (10 min)
- Fig 8: Use setup_waste, run with perp profiler. (40 min)
- Fig 10, left: Run continuous LBench with pcm-numa. (1 min)
- Fig 10, middle: Run continuous LBench with pcm-numa, and also constant LBench. (2 min)
- Fig 10, right: Use setup_waste, then run constant LBench in a loop during workload execution. (20 min)
- Fig 9: Use setup_waste, then run continuous LBench during workload execution. (60 min)
- Fig 11: Use setup_waste, then run with perp profiler (left/middle), and constant LBench (right). (10 min)
- Fig 12: Use setup_waste and LBench as described in the paper. Run 100 executions of each configuration. (40 h)

The results are expected to match the results presented in the paper. The paper also includes an extensive description and evaluation of these results.