Though large-scale cluster systems remain the dominant solution for request and data-level parallelism [12], there have been a resurgence towards applying HPC techniques (e.g., DSM) for more efficient heterogeneous computation with more tightly-coupled heterogeneous nodes providing (hardware) acceleration for one another [7] [ADD MORE CITATIONS] Within the scope of one node, *heterogeneous memory management (HMM)* enables the use of OS-controlled, unified memory view into the entire memory landscape across attached devices [11], all while using the same libc function calls as one would with SMP programming, the underlying complexities of memory ownership and locality managed by the OS kernel.

Nevertheless, while HMM promises a distributed shared memory approach towards exposing CPU and peripheral memory, applications (drivers and frontends) that exploit HMM to provide ergonomic programming models remain fragmented and narrowly-focused. Existing efforts in exploiting HMM in Linux predominantly focus on exposing global address space abstraction to GPU memory – a largely non-coordinated effort surrounding both *in-tree* and proprietary code [10, 1]. Limited effort have been done on incorporating HMM into other variants of accelerators in various system topologies.

Orthogonally, allocation of hardware accelerator resources in a cluster computing environment becomes difficult when the required hardware acceleration resources of one workload cannot be easily determined and/or isolated. Within a cluster system there may exist a large amount of general-purpose worker nodes and limited amount of hardware-accelerated nodes. Further, it is possible that every workload performed on this cluster wishes for hardware acceleration from time to time, but never for a relatively long time. Many job scheduling mechanisms within a cluster *move data near computation* by migrating the entire job/container between general-purpose and accelerator nodes [24, 23]. This way of migration naturally incurs large overhead – accelerator nodes which strictly perform in-memory computing without ever needing to touch the container's filesystem should not have to install the entire filesystem locally, for starters. Moreover, must *all* computations be near data? [21], for example, shows that RDMA over fast network interfaces (25×8Gbps) result negligible impact on tail latencies but high impact on throughput when bandwidth is maximized.

This thesis paper builds upon an ongoing research effort in implementing a tightly coupled cluster where HMM abstractions allow for transparent RDMA access from accelerator nodes to local data and data migration near computation, focusing on the effect of replacement policies on balancing the cost between near-data and far-data computation between home node and accelerator node. Specifically, this paper explores the possibility of implementing shared page movement between home and accelerator nodes to enable efficient memory over-commit without the I/O-intensive swapping overhead.

The rest of the chapter is structured as follows...

1 Experiences from Software DSM

The majority of contributions to the study of software DSM systems come from the 1990s [6, 9, 14, 13]. These developments follow from the success of the Stanford DASH project in the late 1980s – a hardware distributed shared memory (i.e., NUMA) implementation of a multiprocessor that first proposed the *directory-based protocol* for cache coherence, which stores the ownership information of cache lines to reduce unnecessary communication that prevented SMP processors from scaling out [18].

While developments in hardware DSM materialized into a universal approach to cache-coherence in contemporary many-core processors (e.g., *Ampere Altra*[2]), software DSMs in clustered computing languished in favor of loosely-coupled nodes performing data-parallel computation, communicating via message-passing. Bandwidth limitations with the network interfaces of the late 1990s was insufficient to support the high traffic incurred by DSM and its programming model [25, 19].

New developments in network interfaces provides much improved bandwidth and latency compared to ethernet in the 1990s. RDMA-capable NICs have been shown to improve the training efficiency sixfold compared to distributed TensorFlow via RPC, scaling positively over non-distributed training [16]. Similar results have been observed for Spark[20] and what?. Consequently, there have been a resurgence of interest in software DSM systems and their corresponding programming models [22, 8].

1.1 Munin: Multiple Consistency Protocols

Munin[9] is one of the older developments in software DSM systems. The authors of Munin identify that *false-sharing*, occurring due to multiple processors writing to different offsets of the same page triggering invalidations, is strongly detrimental to the performance of shared-memory systems. To combat this, Munin exposes annotations as part of its programming model to facilitate multiple consistency protocols on top of release consistency. An immutable shared memory object across readers, for example, can be safely copied without concern for coherence between processors. On the other hand, the *write-shared* annotation explicates that a memory object is written by multiple processors without synchronization – i.e., the programmer guarantees that only false-sharing occurs within this granularity. Annotations such as these explicitly disables subsets of consistency procedures to reduce communication in the network fabric, thereby improving the performance of the DSM system.

Perhaps most importantly, experiences from Munin show that *restricting the flexibility of programming model can lead to more performant coherence models*, as corroborated by the now-foundational *Resilient Distributed Database* paper [27] – which powered many now-popular scalable data processing frameworks such as *Hadoop MapReduce*[3] and *APACHE Spark*[4]. "To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory [based on]... transformations rather than... updates to shared state" [27]. This al-

lows for the use of transformation logs to cheaply synchronize states between unshared address spaces – a much desired property for highly scalable, loosely-coupled clustered systems.

1.2 Treadmarks: Multi-Writer Protocol

Treadmarks[6] is a software DSM developed in 1996

2 HPC and Partitioned Global Address Space

Improvement in NIC bandwidth and transfer rate allows for applications that expose global address space, as well as RDMA technologies that leverage single-writer protocols over hierarchical memory nodes. [**GAS and PGAS (Partitioned GAS) technologies for example Openshmem, OpenMPI, Cray Chapel, etc. that leverage specially-linked memory sections and `/dev/shm` to abstract away RDMA access**].

Contemporary works on DSM systems focus more on leveraging hardware advancements to provide fast and/or seamless software support. Adrias [21], for example, implements a complex system for memory disaggregation over multiple compute nodes connected via the *ThymesisFlow*-based RDMA fabric, where they observed significant performance improvements over existing data-intensive processing frameworks, for example APACHE Spark, Memcached, and Redis, over no-disaggregation (i.e., using node-local memory only, similar to cluster computing) systems.

2.1 Programming Model

2.2 Move Data to Process, or Move Process to Data?

(TBD – The former is costly for data-intensive computation, but the latter may be impossible for certain tasks, and greatly hardens the replacement problem.)

3 Replacement Policy

In general, three variants of replacement strategies have been proposed for either generic cache block replacement problems, or specific use-cases where contextual factors can facilitate more efficient cache resource allocation:

- General-Purpose Replacement Algorithms, for example LRU.
- Cost-Model Analysis
- Probabilistic and Learned Algorithms

3.1 General-Purpose Replacement Algorithms

Practically speaking, in the general case of the cache replacement problem, we desire to predict the re-reference interval of a cache block [15]. This follows from the Belady's algorithm – the optimal case for the *ideal* replacement problem occurs when, at eviction time, the entry with the highest re-reference interval is replaced. Under this framework, therefore, the commonly-used LRU algorithm could be seen as a heuristic where the re-reference interval for each entry is predicted to be immediate. Fortunately, memory access traces of real computer systems agree with this tendency due to spatial locality [**source**]. (Real systems are complex, however, and there are other behaviors...) On the other hand, the hypothetical LFU algorithm is a heuristic that captures frequency. [**. . .**] While the textbook LFU algorithm suffers from needing to maintain a priority-queue for frequency analysis, it was nevertheless useful for keeping recurrent (though non-recent) blocks from being evicted from the cache [**source**].

Derivatives from the LRU algorithm attempts to balance between frequency and recency. [**Talk about LRU-K, LRU-2Q, LRU-MQ, LIRS, ARC here . . .**]

Advancements in parallel/concurrent systems had led to a rediscovery of the benefits of using FIFO-derived replacement policies over their LRU/LFU counterparts, as book-keeping operations on the uniform LRU/LFU state proves to be (1) difficult for synchronization and, relatedly, (2) cache-unfriendly [26]. [**Talk about FIFO, FIFO-CLOCK, FIFO-CAR, FIFO-QuickDemotion, and Dueling CLOCK here . . .**]

Finally, real-life experiences have shown the need to reduce CPU time in practical applications, owing from one simple observation – during the fetch-execution cycle, all processors perform blocking I/O on the memory. A cache-unfriendly design, despite its hypothetical optimality, could nevertheless degrade the performance of a system during low-memory situations. In fact, this proves to be the driving motivation behind Linux's transition away from the old LRU-2Q page replacement algorithm into the more coarse-grained Multi-generation LRU algorithm, which has been mainlined since v6.1.

3.2 Cost-Model Analysis

The ideal case for the replacement problem fails to account for invalidation of cache entries. It also assumes for a uniform, dual-hierarchical cache-store model that is insufficient to capture the heterogeneity of today's massively-parallel, distributed systems. High-speed network interfaces are capable of exposing RDMA interfaces between computer nodes, which amount to almost twice as fast RDMA transfer when compared to swapping over the kernel I/O stack, while software that bypass the kernel I/O stack is capable of stretching the bandwidth advantage even more (source). This creates an interesting network topology between RDMA-enabled nodes, where, in addition to swapping at low-memory situations, the node may opt to "swap" or simply drop the physical page in order to lessen the cost of page misses.

[**Talk about GreedyDual, GDSF, BCL, Amortization**]

Traditionally, replacement policies based on cost-model analysis were utilized in content-delivery networks, which had different consistency models compared to finer-grained systems. HTTP servers need not pertain to strong consistency models, as out-of-date information is considered permissible, and single-writer scenarios are common. Consequently, most replacement policies for static content servers, while making strong distinction towards network topology, fails to concern for the cases where an entry might become invalidated, let along multi-writer protocols. One early paper [17] examines the efficacy of using page fault frequency as an indicator of preference towards working set inclusion (which I personally think is highly flawed – to be explained). Another paper [5] explores the possibility of taking page fault into consideration for eviction, but fails to go beyond the obvious implication that pages that have been faulted *must* be evicted.

The concept of cost models for RDMA and NUMA systems are relatively underdeveloped, too. (Expand)

3.3 Probabilistic and Learned Algorithms for Cache Replacement

Finally, machine learning techniques and low-cost probabilistic approaches have been applied on the ideal cache replacement problem with some level of success. [**Talk about LeCaR, CACHEUS here**].

4 Cache Coherence and Consistency in DSM Systems

(I need to read more into this. Most of the contribution comes from CPU caches, less so for DSM systems.) [**Talk about JIAJIA and Treadmark's coherence protocol.**]

Consistency and communication protocols naturally affect the cost for each faulted memory access ...

[**Talk about directory, transactional, scope, and library cache coherence, which allow for multi-casted communications at page fault but all with different levels of book-keeping.**]

References

[1] URL: `https://www.phoronix.com/search/Heterogeneous%20Memory%20Management`.

[2] URL: `https://uawartifacts.blob.core.windows.net/upload-files/Altra_Max_Rev_A1_DS_v1_15_20230809_b7cdce449e_424d129849.pdf`.

[3] URL: `https://hadoop.apache.org/`.

[4] URL: https://spark.apache.org/.

[5] J. Aguilar and E.L. Leiss. "A Coherence-Replacement Protocol For Web Proxy Cache Systems". In: *International Journal of Computers and Applications* 28.1 (2006), pp. 12–18. DOI: 10.1080/1206212X.2006.11441783. eprint: https://doi.org/10.1080/1206212X.2006.11441783. URL: https://doi.org/10.1080/1206212X.2006.11441783.

[6] Cristiana Amza et al. "Treadmarks: Shared memory computing on networks of workstations". In: *Computer* 29.2 (1996), pp. 18–28.

[7] Javier Cabezas et al. "GPU-SM: shared memory multi-GPU programming". In: *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. 2015, pp. 13–24.

[8] Qingchao Cai et al. "Efficient distributed memory management with RDMA and caching". In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1604–1617.

[9] John B Carter, John K Bennett, and Willy Zwaenepoel. "Implementation and performance of Munin". In: *ACM SIGOPS Operating Systems Review* 25.5 (1991), pp. 152–164.

[10] Jonathan Corbet. *Heterogeneous memory management meets EXPORT_SYMBOL_GPL()*. 2018. URL: https://lwn.net/Articles/757124/.

[11] Mark Harris. *Unified memory for cuda beginners*. 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[13] Weiwu Hu, Weisong Shi, and Zhimin Tang. "JIAJIA: A software DSM system based on a new cache coherence protocol". In: *High-Performance Computing and Networking: 7th International Conference, HPCN Europe 1999 Amsterdam, The Netherlands, April 12–14, 1999 Proceedings 7*. Springer. 1999, pp. 461–472.

[14] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. "Thread migration and its applications in distributed shared memory systems". In: *Journal of Systems and Software* 42.1 (1998), pp. 71–87.

[15] Aamer Jaleel et al. "High performance cache replacement using re-reference interval prediction (RRIP)". In: *ACM SIGARCH computer architecture news* 38.3 (2010), pp. 60–71.

[16] Chengfan Jia et al. "Improving the performance of distributed tensorflow with RDMA". In: *International Journal of Parallel Programming* 46 (2018), pp. 674–685.

[17] Richard P. LaRowe and Carla Schlatter Ellis. "Page placement policies for NUMA multiprocessors". In: *Journal of Parallel and Distributed Computing* 11.2 (1991), pp. 112–129. ISSN: 0743-7315. DOI: https://doi.org/10.1016/0743-7315(91)90117-R. URL: https://www.sciencedirect.com/science/article/pii/074373159190117R.

[18] Daniel Lenoski et al. “The stanford dash multiprocessor”. In: *Computer* 25.3 (1992), pp. 63–79.

[19] Honghui Lu et al. “Message passing versus distributed shared memory on networks of workstations”. In: *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. IEEE. 1995, pp. 37–37.

[20] Xiaoyi Lu et al. “Accelerating spark with RDMA for big data processing: Early experiences”. In: *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*. IEEE. 2014, pp. 9–16.

[21] Dimosthenis Masouros et al. “Adrias: Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 855–869.

[22] Jacob Nelson et al. “{Latency-Tolerant} software distributed shared memory”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 291–305.

[23] SeungYong Oh and JongWon Kim. “Stateful Container Migration employing Checkpoint-based Restoration for Orchestrated Container Clusters”. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. 2018, pp. 25–30. DOI: 10.1109/ICTC.2018.8539562.

[24] Manuel Rodríguez-Pascual et al. “Job migration in hpc clusters by means of checkpoint/restart”. In: *The Journal of Supercomputing* 75 (2019), pp. 6517–6541.

[25] Paul Werstein, Mark Pethick, and Zhiyi Huang. “A performance comparison of dsm, pvm, and mpi”. In: *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE. 2003, pp. 476–482.

[26] Juncheng Yang et al. “FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion”. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 2023, pp. 70–79.

[27] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.