Though large-scale cluster systems remain the dominant solution for request and data-level parallelism [15], there have been a resurgence towards applying HPC techniques (e.g., DSM) for more efficient heterogeneous computation with tighter-coupled heterogeneous nodes providing (hardware) acceleration for one another [8, 26, 21] Orthogonally, within the scope of one motherboard, *heterogeneous memory management (HMM)* enables the use of OS-controlled, unified memory view across both main memory and device memory [14], all while using the same libc function calls as one would with SMP programming, the underlying complexities of memory ownership and data placement automatically managed by the OS kernel. On the other hand, while HMM promises a distributed shared memory approach towards exposing CPU and peripheral memory, applications (drivers and front-ends) 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 [11, 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 accelerator resources of one workload cannot be easily determined and/or isolated as a "stage" of computation. 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 asks 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 [35, 31]. This way of migration naturally incurs large overhead – accelerator nodes which strictly perform computation on data in memory 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 performed near data? [28], for example, shows that RDMA over fast network interfaces (25 Gbps × 8), when compared to node-local setups, result in 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 migration of data near computation, leveraging different consistency model and coherency protocols to amortize the communication cost for shared data. More specifically, this thesis explores the following:

- The effect of cache coherency maintenance, specifically OS-initiated, on RDMA programs.
- Implementation of cache coherency in cache-incoherent kernel-side RDMA clients.

- Discussion of memory models and coherence protocol designs for a single-writer, multi-reader RDMA-based DSM system.

The rest of the chapter is structured as follows:

- We identify and discuss notable developments in software-implemented DSM systems, and thus identify key features of contemporary advancements in DSM techniques that differentiate them from their predecessors.
- We identify alternative (shared memory) programming paradigms and compare them with DSM, which sought to provide transparent shared address space among participating nodes.
- We give an overview of coherency protocol and consistency models for multi-sharer DSM systems.
- We provide a primer to cache coherency in ARM64 systems, which *do not* guarantee cache-coherent DMA, as opposed to x86 systems [39].

1 Experiences from Software DSM

A majority of contributions to software DSM systems come from the 1990s [5, 10, 18, 17]. These developments follow from the success of the Stanford DASH project in the late 1980s – a hardware distributed shared memory (specifically 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 previous multiprocessors from scaling out [22].

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 [40, 24].

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 [19]. Similar results have been observed for APACHE Spark [25] and SMBDirect [23]. Consequently, there have been a resurgence of interest in software DSM systems and programming models [30, 9].

1.1 Munin: Multi-Consistency Protocol

Munin[10] 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 exhibited by the now-foundational *Resilient Distributed Database* paper [41] 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" [41]. This allows 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[5] is a software DSM system developed in 1996, which featured an intricate *interval*-based multi-writer protocol that allows multiple nodes to write to the same page without false-sharing. The system follows a release-consistent memory model, which requires the use of either locks (via `acquire`, `release`) or barriers (via `barrier`) to synchronize. Each *interval* represents a time period in-between page creation, `release` to another processor, or a `barrier`; they also each correspond to a *write notice*, which are used for page invalidation. Each `acquire` message is sent to the statically-assigned lock-manager node, which forwards the message to the last releaser. The last releaser computes the outstanding write notices and piggy-backs them back for the acquirer to invalidate its own cached page entry, thus signifying entry into the critical section. Consistency information, including write notices, intervals, and page diffs, are routinely garbage-collected which forces cached pages in each node to become validated.

Compared to *Treadmarks*, the system described in this paper uses a single-writer protocol, thus eliminating the concept of "intervals" – with regards to synchronization, each page can be either in-sync (in which case they can be safely shared) or out-of-sync (in which case they must be invalidated/updated). This comes with the following advantage:

- Less metadata for consistency-keeping.
- More adherent to the CPU-accelerator dichotomy model.

- Much simpler coherence protocol, which reduces communication cost.

In view of the (still) disparate throughput and latency differences between local and remote memory access [9], the simpler coherence protocol of single-writer protocol should provide better performance on the critical paths of remote memory access.

1.3 Hotpot: Single-Writer & Data Replication

Newer works such as *Hotpot*[36] apply distributed shared memory techniques on persistent memory to provide "transparent memory accesses, data persistence, data reliability, and high availability". Leveraging on persistent memory devices allow DSM applications to bypass checkpoints to block device storage [36], ensuring both distributed cache coherence and data reliability at the same time [36].

We specifically discuss the single-writer portion of its coherence protocol. The data reliability guarantees proposed by the *Hotpot* system requires each shared page to be replicated to some *degree of replication.* Nodes who always store latest replication of shared pages are referred to as "owner nodes", which arbitrate other nodes to store more replications in order to reach the degree of replication quota. At acquisition time, the acquiring node asks the access-management node for single-writer access to shared page, who grants it if no other critical section exists, alongside list of current owner nodes. At release time, the releaser first commits its changes to all owner nodes which, in turn, commits its received changes across lesser sharers to achieve the required degree of replication. These two operations are all acknowledged back in reverse order. Once all acknowledgements are received from owner nodes by commit node, the releaser tells them to delete their commit logs and, finally, tells the manager node to exit critical section.

The required degree of replication and logged commit transaction until explicit deletion facilitate crash recovery at the expense of worse performance over release-time I/O. While the study of crash recovery with respect to shared memory systems is out of the scope of this thesis, this paper provides a good framework for a **correct** coherence protocol for a single-writer, multiple-reader shared memory system, particularly when the protocol needs to cater for a great variety of nodes each with their own memory preferences (e.g., write-update vs. write-invalidate, prefetching, etc.).

1.4 MENPS: A Return to DSM

MENPS[13] leverages new RDMA-capable interconnects as a proof-of-concept that DSM systems and programming models can be as efficient as *partitioned global address space* (PGAS) using today's network interfaces. It builds upon *TreadMark*'s [5] coherence protocol and crucially alters it to a *floating home-based* protocol, based on the insight that diff-transfers across the network is comparatively costly compared to RDMA intrinsics – which implies preference towards local diff-merging. The home node then acts as the data supplier for every shared page within the system.

Compared to PGAS frameworks (e.g., MPI), experimentation over a subset of *NAS Parallel Benchmarks* shows that MENPS can obtain comparable speedup in some of the computation tasks, while achieving much better productivity due to DSM's support for transparent caching, etc. [13]. These results back up their claim that DSM systems are at least as viable as traditional PGAS/message-passing frameworks for scientific computing, also corroborated by the resurgence of DSM studies later on[28].

2 PGAS and Message Passing

While the feasibility of transparent DSM systems over multiple machines on the network has been made apparent since the 1980s, predominant approaches to "scaling-out" programs over the network relies on the message-passing approach [38]. The reasons are twofold:

1. Programmers would rather resort to more intricate, more predictable approaches to scaling-out programs over the network [38]. This implies manual/controlled data sharding over nodes, separation of compute and communication "stages" of computation, etc., which benefit performance analysis and engineering.

2. Enterprise applications value throughput and uptime of relatively computationally inexpensive tasks/resources [15], which requires easy scalability of tried-and-true, latency-inexpensive applications. Studies in transparent DSM systems mostly require exotic, specifically-written programs to exploit global address space, which is fundamentally at odds in terms of reusability and flexibility required.

2.1 PGAS

Partitioned Global Address Space (PGAS) is a parallel programming model that (1) exposes a global address space to all machines within a network and (2) explicates distinction between local and remote memory [12]. Oftentimes, message-passing frameworks, for example *OpenMPI*, *OpenFabrics*, and *UCX*, are used as backends to provide the PGAS model over various network interfaces/platforms (e.g., Ethernet and Infiniband)[37, 33].

Notably, implementation of a *global* address space across machines on top of machines already equipped with their own *local* address space (e.g., cluster nodes running commercial Linux) necessitates a global addressing mechanism for shared/shared data objects. DART[42], for example, utilizes a 128-bit "global pointer" to encode global memory object/segment ID and access flags in the upper 64 bits and virtual addresses in the lower 64 bits for each (slice of) memory object allocated within the PGAS model. A *non-collective* PGAS object is allocated entirely local to the allocating node's memory, but registered globally. Consequently, a single global pointer is recorded in the runtime with corresponding permission flags for the context of some user-defined group of associated nodes. Comparatively, a *collective* PGAS object is allocated such that a partition of the object (i.e., a sub-array of the repr) is stored in each of the associated node – for a k-partitioned object, k global pointers are recorded in the runtime each pointing to the same object, with different offsets and (intuitively) independently-chosen virtual addresses. Note that this design naturally requires virtual addresses within each node to be *pinned* – the allocated object cannot be re-addressed to a different virtual address, thus preventing the global pointer that records the local virtual address from becoming spontaneously invalidated.

Similar schemes can be observed in other PGAS backends/runtimes, albeit they may opt to use a map-like data structure for addressing instead. In general, despite both PGAS and DSM systems provide memory management over remote nodes, PGAS frameworks provide no transparent caching and transfer of remote memory objects accessed by local nodes. The programmer is still expected to handle data/thread movement manually when working with shared memory over network to maximize their performance metrics of interest.

2.2 Message Passing

Message Passing remains the predominant programming model for parallelism between loosely-coupled nodes within a computer system, much as it is ubiquitous in supporting all levels of abstraction within any concurrent components of a computer system. Specific to cluster computing systems is the message-passing programming model, where parallel programs (or instances of the same parallel program) on different nodes within the system communicate via exchanging messages over network between these nodes. Such models exchange programming model productivity for more fine-grained control over the messages passed, as well as more explicit separation between communication and computation stages within a programming subproblem.

Commonly, message-passing backends function as *middlewares* – communication runtimes – to aid distributed software development [38]. Such a message-passing backend expose facilities for inter-application communication to frontend developers while transparently providing security, accounting, and fault-tolerance, much like how an operating system may provide resource management, scheduling, and security to traditional applications [38]. This is the case for implementing the PGAS programming model, which mostly rely on common message-passing backends to facilitate orchestrated data manipulation across distributed nodes. Likewise, message-passing backends, including RDMA API, form the backbone of many research-oriented DSM systems [13, 16, 9, 20].

Message-passing between network-connected nodes may be *two-sided* or *one-sided*. The former models an intuitive workflow to sending and receiving datagrams over the network – the sender initiates a transfer; the receiver copies a received packet from the network card into a kernel buffer; the receiver's kernel filters the packet and (optionally) [34] copies the internal message into the message-passing runtime/middleware's address space; the receiver's middleware inspects the copied message and performs some procedures accordingly, likely also involving copying slices of message data to some registered distributed shared memory buffer for the distributed application to access. Despite it being a highly intuitive model of data manipulation over the network, this poses a fundamental performance issue: because the process requires the receiver's kernel AND userspace to exert CPU-time, upon reception of each message, the receiver node needs to proactively exert CPU-time to move the received data from bytes read from NIC devices to userspace. Because this happens concurrently with other kernel and userspace routines in a concurrent system, a preemptable kernel may incur significant latency if the kernel routine for packet filtering is pre-empted by another kernel routine, userspace, or IRQs.

Comparatively, a "one-sided" message-passing scheme, for example RDMA, allows the network interface card to bypass in-kernel packet filters and perform DMA on registered memory regions. The NIC can hence notify the CPU via interrupts, thus allowing the kernel and the userspace programs to perform callbacks at reception time with reduced latency. Because of this advantage, many recent studies attempt to leverage RDMA APIs for improved distributed data workloads and creating DSM middlewares [25, 19, 13, 16, 9, 20].

3 Consistency Model and Cache Coherence

Consistency model specifies a contract on allowed behaviors of multi-processing programs with regards to a shared memory [29]. One obvious conflict, which consistency models aim to resolve, lies within the interaction between processor-native programs and multi-processors, all of whom needs to operate on a shared memory with heterogeneous cache topologies. Here, a well-defined consistency model aims to resolve the conflict on an architectural scope. Beyond consistency models for bare-metal systems, programming languages [7, 6, 27, 32] and paradigms [5, 16, 9] define consistency models for parallel access to shared memory on top of program order guarantees to explicate program behavior under shared memory parallel programming across underlying implementations.

3.1 Consistency Model in DSM

3.2 Coherence Protocol

3.3 DMA and Cache Coherence

3.4 Cache Coherence in ARMv8

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

[6] Hans J Boehm and Lawrence Crowl. *C++ Atomic Types and Operations.* 2007. URL: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html`.

[7] *BS ISO/IEC 9899:2011: Information technology. Programming languages. C.* eng. 2013.

[8] 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.

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

[10] 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.

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

[12] Mattias De Wael et al. "Partitioned global address space languages". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–27.

[13] Wataru Endo, Shigeyuki Sato, and Kenjiro Taura. "MENPS: a decentralized distributed shared memory exploiting RDMA". In: *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM).* IEEE. 2020, pp. 9–16.

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

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

[16] Yang Hong et al. “Scaling out NUMA-aware applications with RDMA-based distributed shared memory”. In: *Journal of Computer Science and Technology* 34 (2019), pp. 94–112.

[17] 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.

[18] 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.

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

[20] Stefanos Kaxiras et al. “Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 3–14. ISBN: 9781450335508. DOI: `10.1145/2749246.2749250`. URL: `https://doi.org/10.1145/2749246.2749250`.

[21] Ahmed Khawaja et al. “Sharing, Protection, and Compatibility for Reconfigurable Fabric with {AmorphOS}”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 107–127.

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

[23] Feng Li et al. “Accelerating relational databases by leveraging remote memory and RDMA”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 355–370.

[24] 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.

[25] 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.

[26] Jiacheng Ma et al. “A hypervisor for shared-memory FPGA platforms”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 827–844.

[27] Jeremy Manson and Brian Goetz. 2004. URL: `https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`.

[28] 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.

[29] Vijay Nagarajan et al. *A primer on memory consistency and cache coherence*. Springer Nature, 2020.

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

[31] 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`.

[32] *Ordering in core::sync::atomic - Rust*. 2024. URL: `https://doc.rust-lang.org/core/sync/atomic/enum.Ordering.html`.

[33] *Platform-Specifc Notes*. 2023. URL: `https://chapel-lang.org/docs/platforms/index.html#`.

[34] The FreeBSD Project. *FreeBSD manual pages*. 2021. URL: `https://man.freebsd.org/cgi/man.cgi?query=bpf&manpath=FreeBSD+14.0-RELEASE+and+Ports`.

[35] 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.

[36] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. "Distributed Shared Persistent Memory". In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, pp. 323–337. ISBN: 9781450350280. DOI: `10.1145/3127479.3128610`. URL: `https://doi.org/10.1145/3127479.3128610`.

[37] *upcc.1*. 2022. URL: `https://upc.lbl.gov/docs/user/upcc.html`.

[38] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[39] Arjan van de Ven. *Background on ioremap, cacheing, cache coherency on x86*. 2008. URL: `https://lkml.org/lkml/2008/4/29/480`.

[40] 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.

[41] 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`.

[42] Huan Zhou et al. "DART-MPI: An MPI-based implementation of a PGAS runtime system". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 2014, pp. 1–11.