Evaluating Hardware Memory Disaggregation under Delay and Contention

Archit Patke*, Haoran Qiu*, Saurabh Jha‡, Srikumar Venugopal†, Michele Gazzetti†, Christian Pinto†, Zbigniew Kalbarczyk*, Ravishankar Iyer *

*University of Illinois at Urbana-Champaign, USA
‡IBM Research, USA
†IBM Research, Ireland

***Abstract*—Hardware memory disaggregation is an emerging trend in datacenters that provides access to remote memory as part of a shared pool or unused memory on machines across the network. Memory disaggregation aims to improve memory utilization and scale memory-intensive applications. Current state-of-the-art prototypes have shown that hardware disaggregated memory is a reality at the rack-scale. However, the memory utilization benefits of memory disaggregation can only be fully realized at larger scales enabled by a datacenter-wide network. Introduction of a datacenter network results in new performance and reliability failures that may manifest as higher network latency. Additionally, sharing of the network introduces new points of contention between multiple applications. In this work, we characterize the impact of variable network latency and contention in an open-source hardware disaggregated memory prototype — ThymesisFlow. To support our characterization, we have developed a *delay injection framework* that introduces delays in remote memory access to emulate network latency. Based on the characterization results, we develop insights into how reliability and resource allocation mechanisms should evolve to support hardware memory disaggregation beyond rack-scale in datacenters.**

I. Introduction

Hardware memory disaggregation [1]–[3] is an emerging trend in datacenters that provides access to remote memory as part of a shared pool, or by borrowing unused memory from remote machines across the network. To enable memory accesses across the network, processor cache misses are re-directed on a cache-coherent interconnect [4]–[6] to a special remote memory controller that further forwards the cache misses on the network. Memory disaggregation has multiple advantages as it can potentially 1) increase utilization of data center memory by upto 30–40% [7], and 2) scale memory-intensive applications (such as in-memory databases [8], parallel data processing frameworks [9], and HPC applications [10]) by provisioning them with additional memory in a cost-effective manner.

Current state-of-the-art prototypes [1] have shown that hardware disaggregated memory is a reality at the rack-scale [1]. However, scaling disaggregation further to fully realize its benefits can be challenging because of higher and dynamic memory access latency imposed by the datacenter network.

[1]We consider the memory borrowing model for disaggregation, where compute nodes can borrow unutilized memory from other datacenter nodes. An alternative model is memory pooling where nodes access CPU-less memory devices over the network (as discussed in §V)

In this work, we characterize the impact of contention and variable network latency in a rack-scale hardware disaggregated memory prototype — ThymesisFlow [1]. To support our characterization, we have developed a *delay injector* that delays outgoing remote memory requests to emulate delays associated with the network. Injecting delay is a useful approach to assess a system [11], [12] as delays can arise due to multiple performance (such as network congestion) and reliability (such as link repair) failures. Our delay injection experiments help characterize the performance and reliability properties of a hardware disaggregated memory system. We extrapolate the characterization to suggest future research directions for the evolution of resource management and reliability mechanisms in hardware disaggregated memory beyond the rack-scale.

Our key findings and insights are following:

- *Current processor architecture is resilient to timeout-induced failures introduced by tail network latency.* While very high network delay leads to OS-level timeouts and system crashes, we find that such delay is beyond the tail latency observed in current datacenter network fabrics. For example, we find that introducing additional network delay which corresponds to the 99th percentile datacenter network latency [13] to a hardware disaggregated memory system, leads to performance degradation but does not cause any system crashes. Therefore, improving processor resilience to delay-induced failures is not of immediate concern for beyond rack-scale memory disaggregation.
- *Resource management mechanisms for disaggregated memory need to enable Quality-of-Service (QoS) features.* Impact of network delay on application-level metrics (such as request completion rate and job completion times) is drastically different across workloads. For example, a delay injection of 30 μs leads to a performance degradation of less than 1% in Redis, an in-memory key-value store, but an identical delay injection in the Graph 500 benchmark leads to a 7× job completion time increase. As the impact of additional delay is substantially different for different applications, future resource allocation mechanisms need to enable Quality-

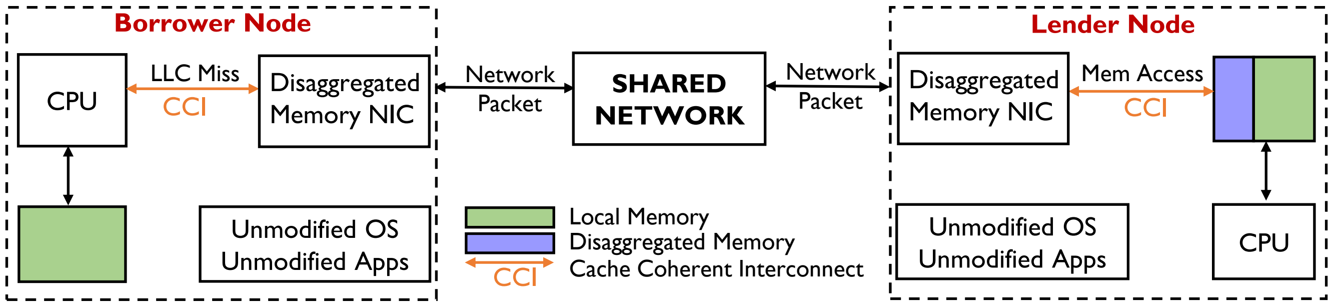

Figure 1: Overview of a remote memory access in a hardware disaggregated memory borrowing system.

of-Service (QoS) features to support workloads that are sensitive to memory access latency increase. Examples of such resource allocation mechanisms include page migration at the operating system, congestion control and packet scheduling at the network, and remote memory allocations at the control plane.

- *Impact of network latency dominates over memory contention at lender node.* Multiple concurrently running applications at the lender node do not lead to a significant performance impact on disaggregated memory at the borrower node. As memory bus bandwidth is typically much higher than network bandwidth, the network continues to remain the bottleneck even when memory is shared. These experimental insights can be used to improve the dynamic assignment of disaggregated memory. For example, a lender node with multiple running applications and an idle lender node can be equally viable candidates for remote memory reservation and access.

II. Background

Several academic and industry prototypes have been proposed to enable memory disaggregation by making changes to the application [14], operating system [15], and hardware [1]. While each approach comes with its own advantages, we focus on a hardware-based approach for memory disaggregation as it has the lowest memory access latency and does not need modifications to applications and the operating system kernel.

A. A model for hardware memory disaggregation

State-of-the-art architectures [1], [2] for hardware memory disaggregation in the datacenter have proposed using cache lines as basic blocks for remote memory accesses. Using cache-line-sized memory accesses has gained traction as it allows finer-grained access to memory (compared to pages) and avoids expensive page faults. Enabling transfer via cache lines involves: 1) Reserving memory at a datacenter node for remote access, and 2) Enabling processor cache miss redirection to access the reserved remote memory. Decisions involved in reserving memory and configuring access to remote memory are performed by a control plane.

Remote Memory Reservation: Each node in the system is designated a role of either “borrower” (borrowing memory) or “lender” (lending memory) node by the control plane. Role assignment is dynamic and dependent on real-time memory availability and demand of each node across the datacenter. Additionally, the control plane decides the size of memory reservations at each lender node.

Remote Memory Access: Figure 1 shows a representation of a remote memory access from the borrower node CPU (left) to memory at the lender node (right). We describe the memory access path in further detail. Any remote memory address accessed by the borrower node CPU that is not present in the last-level cache is sent to a custom *disaggregated memory NIC* via a cache-coherent interconnect protocol such as CXL [4] or OpenCAPI [6]. The disaggregated memory NIC is implemented via a custom ASIC or FPGA that transforms the cache miss into a network packet by encapsulating with a packet header for network transmission (such as the destination network address, checksum, etc.). Additionally, address translation is implemented to convert addresses at the borrower node to corresponding addresses at the lender node. The network packet containing the cache miss is then transmitted on a network shared between multiple borrower-lender node pairs and can include intermediate switches to support a large-scale datacenter. The network packet reaches the lender node at the disaggregated memory NIC that extracts out the address information. The address is accessed from the local memory of the lender node again via a cache-coherence protocol. The corresponding data is then returned to the borrower node by reversing the entire path. Note that the operating system kernel and applications at the borrower and lender node do not need to be modified to access disaggregated memory.

B. Challenges in scaling current architecture

State-of-the-art prototypes for hardware memory disaggregation are limited to the rack-scale as they use direct point-to-point links (instead of a shared network as described in §II-A) between borrower and lender nodes [1]. Scaling beyond the rack-scale can be advantageous as it can help fully realize the benefits of memory disaggregation such as further improved memory utilization. Attempts are being made to enable the transition beyond rack-scale with the introduction of switching support such as the CXL interconnect [4] and the Gen-Z consortium [16]. The CXL standard provides a cache-coherent interconnect between CPU and accelerators/memory, while the GenZ standard provides a memory-centric server interconnect across nodes. The recent merging of the Gen-Z specification into CXL [17] has sent a clear sign of disaggregation of memory being pushed beyond the limits of the single machine and rack. However, the introduction of a switched network brings additional challenges due to new failure modes (such as network congestion), some of which manifest as increased network latency [18]. Additionally, the use of a large-scale and shared disaggregated memory system introduces resource contention due to multi-tenancy (i.e. multiple applications from different nodes competing for the same lender node).

C. Addressing the Gap

In this work, we characterize a rack-scale hardware disaggregated memory prototype [1] under variable network latency and different contention scenarios to understand the behaviour of a beyond rack-scale hardware disaggregated memory system with a switched network. To emulate higher network latency, we have developed a delay injection framework that adds specified delay values to remote memory access (as described in §III). To create contention, we vary concurrency (i.e. number of applications running simultaneously) at both lender and borrower nodes (as described in §IV-E). Our characterization results provide insights for designing future resource allocation and reliability mechanisms in beyond rack-scale disaggregation with a switched network.

III. Delay Injection

In this section, we describe ThymesisFlow, the hardware disaggregated memory system that we use as a characterization testbed. We also discuss implementation details of the delay injection framework that we use to synthetically emulate higher network latency.

A. Prototype System

We use ThymesisFlow [1], an open-source prototype for hardware memory disaggregation. ThymesisFlow implements a hardware-software co-designed memory disaggregation interconnect on top of the POWER9™ architecture, by directly interfacing the memory bus via the OpenCAPI cache coherence protocol. The prototype is composed of two IBM Power System AC922 nodes. Each node features a dual socket POWER9™ CPU (32 physical cores and 128 parallel hardware threads) and 512GB of RAM. Both nodes are equipped with an AlphaData 9V3 card that features a Xilinx Ultrascale FPGA that implements the OpenCAPI stack and the ThymesisFlow interconnect. Our prototype implements a two-node version of the disaggregated memory model described in §II-A where: 1) the shared network is replaced by a 100Gb/s point-to-point connection over a copper cable, and 2) the disaggregated memory NIC is deployed on the AlphaData 9V3 cards. On the software side, we rely on *libthymesisflow*, a user-space library part of the ThymesysFlow project that configures the FPGAs, and takes care of reserving the memory at the lender node and hot-plugging it to the borrower node. The memory borrowed is dedicated to the borrower node, and it is not accessed from any process running on the lender node.

B. Delay Injection Implementation

We have created a delay injection framework that synthetically emulates the delay between consecutive memory requests. To synthetically generate delays, we introduce an additional module between the routing and multiplexer modules at the compute node egress in the ThymesisFlow prototype. In the context of Figure 1, the delay injection module is part of the borrower's disaggregated memory NIC. The module introduces delay while obeying the AXI4-Stream [19] protocol conventions, used to interconnect the internal blocks of the ThymesisFlow hardware design. The AXI4-Stream data transfer is based on a two-way handshake mechanism of `VALID` and `READY` binary signals that indicate whether data is available and can be processed by the module downstream. Both `READY` and `VALID` signals need to be high for the data to be read and further processed. Our key modification was to keep the `VALID` signal unchanged and set the modified $\texttt{READY}_{\texttt{NEW}}$ signal to be high once every `PERIOD` FPGA clock cycles obeying (1). In (1), `COUNTER` is the number of FPGA clock cycles that have elapsed since system start and $\texttt{READY}_{\texttt{OLD}}$ is the original unmodified `READY` signal.

$$\texttt{READY}_{\texttt{NEW}} = \texttt{READY}_{\texttt{OLD}} \,\&\, \texttt{(COUNTER\%PERIOD==0)} \quad (1)$$

Effectively, a transaction is allowed to proceed once every `PERIOD` cycles if $\texttt{READY}_{\texttt{OLD}}$ and `VALID` signals remain high.

We test our delay injection framework along with the STREAM benchmark at varying values of `PERIOD`. Using our delay injections, we: 1) validate that our injection framework is able to emulate network latency at levels observed in datacenter network fabrics, 2) find strong linear correlation between `PERIOD` and application-level latency measurements, and 3) observe that the overall system continues to have a constant bandwidth-delay product (BDP) across different delay injections. We elaborate on these results further in §IV.

IV. Evaluation

In our evaluation, we first validate the correctness of our delay injection framework (described in §III) in §IV-B. Next, we use the delay injection framework to assess system resilience and application performance degradation in §IV-C and §IV-D. Additionally, we evaluate the effects of application concurrency (i.e. number of simultaneously running applications) at borrower and lender nodes in §IV-E. Using our results, we present insights for design of resource control and reliability mechanisms in future hardware disaggregated memory systems (at the end of each subsection).

A. Benchmarks

We consider benchmarks that represent applications with diverse memory bandwidth demands and latency requirements. In this section, we describe each of these benchmarks and specific configurations used to execute them in the ThymesisFlow prototype with remote memory access.

STREAM: STREAM [20] is a memory testing benchmark that measures memory access times and bandwidth for commonly used kernels. We configured STREAM to use 10 million array elements, requiring a total memory of 0.2 GiB, which is beyond

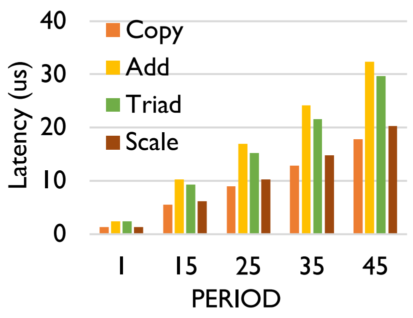

Figure 2: Latency measured by STREAM for varying delay injection.

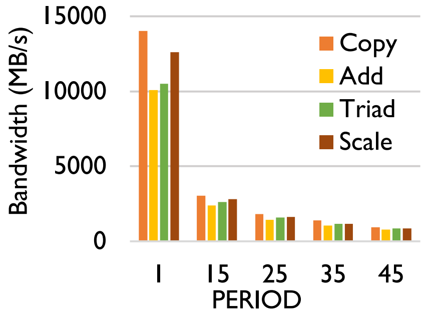

Figure 3: Bandwidth measured by STREAM for varying delay injection.

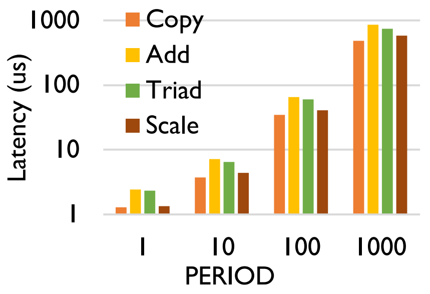

Figure 4: System reliability testing under heavy delay injection.

the total cache size of 120 MiB on each node. Each benchmark run executes four kernels, i.e., "copy", "scale", "add" and "triad". "copy" reads/writes 16 bytes (1 read, 1 write ops) of memory per iteration, performing no floating point operations (FLOPs), "scale" reads/writes the same amount of memory with the same number of operations but it performs 1 FLOP per iteration, "add" accesses 24 bytes of memory (2 read and 1 write ops) and executes 1 FLOP per iteration, and "triad" accesses 24 bytes of memory (2 read and 1 write ops) executing 2 FLOPs per iteration.

Redis: Redis [21] is an in-memory data structure store used as a database, cache, and message broker. To improve performance, Redis works with an in-memory dataset. We test performance of Redis with the Memtier [22] benchmark that generates data for a variety of structures, performs stress testing, and helps understand performance limits. We configured Memtier to use 4 threads with 50 connections per thread that cumulatively send 10000 requests per client. The Memtier benchmark along with Redis uses a working memory set of size ∼4 GB.

Graph 500: The Graph500 benchmark [23] constructs large graphs and performs multiple iterations of Breadth First Search (BFS) and Single Source Shortest Path (SSSP) on a generated graph. We use a problem size of 20 with a edge factor of 16 with the Graph500 benchmark. In this configuration, Graph 500 uses a working memory set of size ∼1 GB.

B. *Validating Delay Injection*

To validate the functionality of the delay injection framework, we run STREAM on the borrower node while keeping the lender node idle, and introduce varying levels of delay via our delay injection framework. Figure 2 shows the variation in measured STREAM latency for varying `PERIOD` values set via the delay injection framework. For the smallest `PERIOD` = 1, the system effectively behaves as the vanilla ThymesisFlow prototype as all valid transactions pass through without any waiting. Other values of `PERIOD` lead to varying levels of delay injection and STREAM-measured latency lies between 1.2–150 μs. The measured range of latency corresponds to the [0-90th]-percentile network latency in production datacenter networks [13], [24]. Thus, our delay injection framework is able to generate sufficiently high latency values to enable realistic emulations of network conditions.

Figure 3 shows variation in measured STREAM bandwidth for varying `PERIOD` values set via the delay injection framework. We find that consumed bandwidth rapidly decreases with additional delay and the bandwidth-delay product remains roughly constant across all the delay injections with a value equal to ∼16.5 kB.

C. *Resilience Assesment*

In this set of experiments, we intend to assess the potential resilience limits of the processor and applications running in the context of a hardware disaggregated memory system. Our experimental setup is identical to the setup described in §IV-B where we run a single STREAM instance on the borrower node while keeping the lender node idle. We use the delay injection framework (described in Section III) to add exponentially increasing levels of delay to stress test our system. Figure 4 shows the impact of increasing latency via increasing `PERIOD` on latency measured by the STREAM benchmark. At `PERIOD` = 1000, STREAM is able to run to completion and measures the average memory access time to be close to 400μs. Despite introducing such high delay, the system stack at the borrower node (including the POWER9™ CPU, OpenCAPI, and the FPGA) continues to remain functional. Thus, we infer that the CPU is resilient to high values of delay. When we increase `PERIOD` further and set it to 10000, the ThymesisFlow compute-side FPGA is no longer detected due to timeout and the disaggregated memory cannot be attached. However, such a high value of `PERIOD` corresponds to a delay of 4 ms, which is far beyond the 99th percentile tail network latency as reported in existing datacenters [13], [24].

	PERIOD=1	PERIOD=1000
Redis	1.01x	1.73x
Graph500 BFS	6x	2209x
Graph500 SSSP	5.3x	1800x

Table I: Impact of high delay on application performance.

We also evaluate the impact of high delay on application performance as shown in Table I. To estimate performance degradation, we use the ratio between the completion time on disaggregated memory under delay injection and the original completion time on local memory. For some applications (Graph 500 and STREAM), we find that extremely high delay leads to severe performance degradation that makes

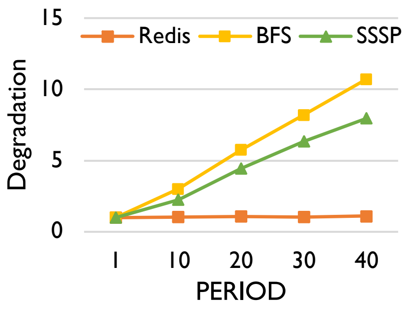

Figure 5: Impact of delay on application performance.

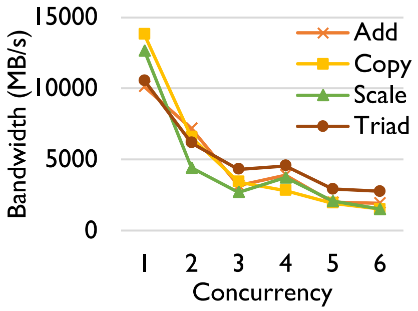

Figure 6: Contention for bandwidth at borrower node.

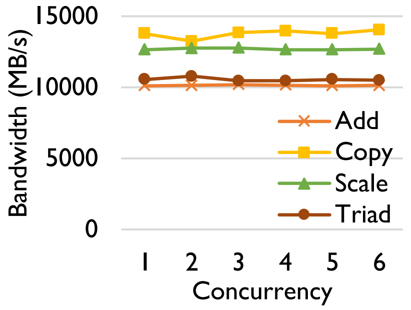

Figure 7: Contention for bandwidth at lender node.

the application completely unresponsive. For example, the performance of Graph 500 degrades by almost 2209× compared to local memory performance. Such an extreme performance hit effectively renders the application unusable and could violate service-level agreements (SLAs) provided by the datacenter operator.

Takeaway: While the CPU is resilient to extremely high delay values, severe performance degradation could violate SLAs for applications.

D. Application Performance Impact

We measure the impact of varying delay on application performance to understand the end-to-end impact of higher network latency. We define application performance on a per-application basis. For example, the number of requests served per second and job completion times are used as metrics to measure the performance of Redis and Graph 500, respectively. To calculate performance degradation, we use the ratio between the degraded runtime due to delay and the original baseline runtime when running on vanilla ThymesisFlow with disaggregated memory. Figure 5 shows the performance degradation due to delay by setting different values of `PERIOD` (as discussed in §III). We observe that application performance degradation can be significantly different for different applications. For example, the performance degradation of Redis is 1.01× at high delay values, effectively amounting to a loss of less than 1%. However, the performance degradation of both Graph 500 benchmarks (BFS and SSSP) is upto 10.7× and 8× , which is significantly higher than Redis. The difference in performance degradation arises because Redis serves requests via the network stack which adds significant serving overhead. While memory access time is also a component of the end-to-end latency, it is negligible compared to the network stack overheads that act as the limiting factor. On the other hand, the Graph 500 benchmarks are almost completely limited by memory and compute, and suffer from much more severe performance degradation.

A hardware disaggregated memory datacenter will run a mix of applications that have different sensitivity to memory access latency. During periods of increased network latency, applications with higher sensitivity to remote memory access latency can benefit from additional resource allocation such as network packet prioritization or page migration to local memory. Therefore, resource allocation mechanisms across the system stack should enable Quality-of-Service (QoS) features to benefit sensitive applications. Examples of such resource control mechanisms include: memory allocation at the control plane, congestion control at the network, and page migration at the operating system.

Takeaway: Application performance degradation under increased remote memory access latency is variable and warrants including QoS features in resource control mechanisms.

E. Resource Contention

We run multiple memory contending applications concurrently to understand the impact of contention on application performance at both the borrower and lender nodes. Specifically, we run multiple instances of the STREAM benchmark in the following two configurations:

- **Memory Contention at the Borrower Node (MCBN):** All instances of STREAM run on the borrower node and use disaggregated memory from the lender node.
- **Memory Contention at the Lender Node (MCLN):** All but one instance of STREAM run on the lender node. A single instance of STREAM runs on the borrower node and uses disaggregated memory from the lender node, thus contending with other STREAM instances running on the lender node.

Figure 6 and Figure 7 show available bandwidth reported by STREAM running at the borrower node in the MCBN and MCLN scenarios respectively. We find that in the case of MCLN, the available bandwidth on the borrower node is independent of the number of concurrent running instances of STREAM. The network bandwidth acts as a greater bottleneck compared to decrease in available memory bus bandwidth at the lender node. Therefore, the impact of memory contention at the lender node does not lead to significant drop in performance at the borrower node. In general, we expect the lender node contention to be insignificant compared to the network as memory bus bandwidth is significantly higher compared to network bandwidth (100s of GB/s vs. 100s of Gb/s).

However, in the case of MCBN, multiple STREAM instances contend on the same borrower node and there is an equal division of bandwidth amongst the competing STREAM instances as they compete for the bottleneck network bandwidth.

The insight that lender-side contention for memory is relatively insignificant can be used to drive better memory allocation decisions at the control plane. For example, a lender node with multiple running applications and an idle lender node can be equally viable candidates for remote memory reservation and access.

Takeaway: Memory allocation mechanisms can be contention-aware to take advantage of variable contention at borrower and lender nodes.

V. Discussion

Limitations of the delay injection framework. Our delay injection framework injects a near-constant delay and does not create variability across memory accesses. However, production datacenter networks have variable latencies with significant variation at short timescales. In this paper, we demonstrate the effects of variation in a coarse-grained manner by changing injected delay values between two application runs. However, injected delay for a single application is kept constant. It would be interesting to explore the impact on application performance when the delay varies at shorter timescales (i.e. within an application run).

Applicability to alternate models of memory disaggregation. An alternate model of hardware memory disaggregation is memory pooling where the dedicated memory is mananged by a controller without any attached CPUs. If disaggregated memory is deployed with memory pools, results presented in §IV-E could be significantly different. For instance, the memory bandwidth contention between multiple applications could be significantly higher depending on the available bandwidth of each memory pool and the bottleneck could shift from the network to the memory pool itself.

Differences between cache-coherence protocols. Our test system ThymesisFlow uses the OpenCAPI protocol to redirect cache misses to the network. A competing cache coherent interconnect for memory disaggregation is the CXL [4] standard which is similar to OpenCAPI but has some major differences. CXL supports native packet switching on the network and hence does not rely on Ethernet or Infiniband networks. Unlike OpenCAPI, CXL works only with physical addresses at the compute and memory nodes. Additional experimentation is required to understand our results in the context of a CXL-based hardware memory disaggregated system.

VI. Related Work

Remote Memory Systems. Works on distributed shared memory [25]–[29] provide shared memory and cache coherence across nodes. In contrast, hardware memory disaggregation leverages local cache coherence within a single host to expose remote memory to applications without any code changes. The availability of low-latency networking has recently made remote memory practical, resulting in a resurgence of research in this area [15], [30]–[33]. However, these works, including disk swapping [34] rely on page faults and page-based tracking, which severely limits their performance. Some remote memory systems [14], [35]–[37] use an object-based interface that avoids the virtual memory subsystem's overhead, but these systems require modifications of the application code. Hardware memory disaggregation avoids virtual memory overhead by using the local cache coherence traffic to access remote data while remaining transparent to applications.

Impact of Memory Latency on Applications. Previous work has explored the impact of memory bandwidth and latency on application performance. Using the right hardware performance counters to analyze the performance of memory subsystem has been studied. For example, in [38] performance counters are identified to measure the usage of available bandwidth and the percentage of cycles consumed by the components in the memory hierarchy. However, the work does not address the measurement of memory request latency. Workload memory characterization has also been studied extensively. For example, a performance model is proposed in [39] to evaluate the workloads' sensitivity towards memory bandwidth and memory access latency. The model focuses on the characterization of different types of workloads in a static offline environment. However, the static evaluation does not account for the complexity of runtime memory subsystem performance that impacts the application performance. An analytical memory model is presented in [40] to predict the performance of a program on different processors. The model uses static analysis based on reuse distances to estimate the memory latencies at different hierarchies of the memory subsystem. However, the static analysis does not account for the varying runtime factors such as interference from other co-located workloads.

Memory Fault Injection. There has been a large body of work that tests the impact of DRAM errors on operating system and application performance [12], [41]. To the best of our knowledge, no work injects latency to memory requests at an order of magnitude close to network latency.

VII. Conclusion and Future Work

In this work, we characterize a hardware memory disaggregated prototype under emulated network delay and performance contention due to multi-tenancy. Our results suggest directions to explore for future reliability and resource management mechanisms in hardware disaggregated memory. As a part of future work, we aim to improve the delay injection framework by enabling injecting delays according to a distribution instead of fixed values. Additionally, we aim to integrate our insights into resource management mechanisms to improve system performance and reliability.

VIII. Acknowledgements

We thank the anonymous reviewers for their valuable comments that improved the paper. This work is partially supported by the National Science Foundation (NSF) under grant No. CCF 20-29049; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

References

[1] "ThymesisFlow Home Page," https://github.com/OpenCAPI/ThymesisFlow/, accessed: 2022-02-08.

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

[3] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," *arXiv preprint arXiv:2108.03492*, 2021.

[4] "Compute Express Link," https://www.computeexpresslink.org/, accessed: 2021-07-24.

[5] "Ccix," https://www.ccixconsortium.com/, accessed: 2021-07-24.

[6] "opencapi," https://opencapi.org/, accessed: 2021-07-24.

[7] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.

[8] Y. Zhang, C. Ruan, C. Li, X. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo *et al.*, "Towards cost-effective and elastic cloud database deployment via memory disaggregation," *Proceedings of the VLDB Endowment*, vol. 14, no. 10, pp. 1900–1912, 2021.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[10] L. Bergstrom, "Measuring numa effects with the stream benchmark," *arXiv preprint arXiv:1103.3225*, 2011.

[11] H. A. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 1993, pp. 208–217.

[12] S. Han, K. G. Shin, and H. A. Rosenberg, "Doctor: An integrated software fault injection environment for distributed real-time systems," in *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE, 1995, pp. 204–213.

[13] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 139–152.

[14] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[15] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "{LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 69–87.

[16] "GenZ consortium," https://genzconsortium.org/, accessed: 2022-02-08.

[17] "CXL Consortium & Gen-Z Consortium Sign Letter of Intent to Advance Interconnect Technology," https://www.computeexpresslink.org/post/exploring-the-future-cxl-consortium-gen-z-consortium, accessed: 2022-02-08.

[18] S. Jha, A. Patke, J. Brandt, A. Gentile, B. Lim, M. Showerman, G. Bauer, L. Kaplan, Z. Kalbarczyk, W. Kramer *et al.*, "Measuring congestion in high-performance datacenter interconnects," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 37–57.

[19] "AMBA 4 AXI4-Stream Protocol Specification," https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol, accessed: 2022-02-10.

[20] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[21] J. Carlson, *Redis in action*. Simon and Schuster, 2013.

[22] "Memtier Redis benchmark," https://github.com/RedisLabs/memtier_benchmark, accessed: 2022-02-10.

[23] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[24] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan *et al.*, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 514–528.

[25] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[26] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, 1990, pp. 168–176.

[27] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[28] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A low overhead, software-only approach for supporting fine-grain shared memory," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996, pp. 174–185.

[29] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, 1994, pp. 297–306.

[30] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati *et al.*, "Remote regions: a simple abstraction for remote memory," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 775–787.

[31] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[32] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[33] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 843–857.

[34] S. F. Kaplan, L. A. McGeoch, and M. F. Cole, "Adaptive caching for demand prepaging," *ACM SIGPLAN Notices*, vol. 38, no. 2 supplement, pp. 114–126, 2002.

[35] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 54–70.

[36] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "{Latency-Tolerant} software distributed shared memory," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 291–305.

[37] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "{AIFM}:{High-Performance},{Application-Integrated} far memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 315–332.

[38] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting memory-boundedness with hardware performance counters," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 27–38.

[39] R. Clapp, M. Dimitrov, K. Kumar, V. Viswanathan, and T. Willhalm, "Quantifying the performance impact of memory latency and bandwidth for big data workloads," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 213–224.

[40] G. Chennupati, N. Santhi, and S. Eidenbenz, "Scalable performance prediction of codes with memory hierarchy and pipelines," in *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2019, pp. 13–24.

[41] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.