Memory-Disaggregated In-Memory Object Store Framework for Big Data Applications

Robin Abrahamse[1] , **Zaid Al-Ars**[1] , **Ákos Hadnagy**[1]
[1]University of Technology Delft

Abstract — **The concept of memory disaggregation has recently been gaining traction in research. With memory disaggregation, data center compute nodes are able to directly access memory on adjacent nodes and can therefore overcome local memory restrictions, introducing a new data management paradigm for distributed computing. This paper proposes and demonstrates a memory disaggregated in-memory object store framework for big data applications by leveraging the newly introduced ThymesisFlow memory disaggregation system. The framework extends the functionality of the existing Apache Arrow Plasma object store framework to distributed systems by enabling clients to easily and efficiently produce and consume data objects across multiple compute nodes. This allows big data applications to increasingly leverage parallel processing at reduced development cost. In addition, the paper includes latency and throughput measurements to evaluate the framework's performance and to guide the design of future systems that leverage memory disaggregation as well as the newly presented framework.**

1 Introduction

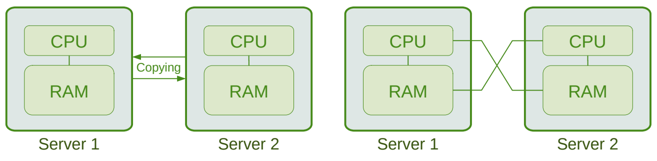
The ongoing expansion in scale of big data workloads demands data centers to increasingly supply higher performance facilities. As progress in power efficiency of the underlying technology lags behind, the power consumption of data centers increases accordingly. It is expected that data center electricity consumption will account for up to 13% of total global electricity supply as soon as 2030 [1]. This puts significant pressure on global energy resources. Improving the efficiency of big data infrastructures is an essential method to ensure the sustainability of data center utilization and limit its environmental impact.

1.1 Memory Disaggregation

Recently, work has been done on behalf of IBM, introducing a memory disaggregation framework for data center infrastructures, called ThymesisFlow [2]. Memory disaggregation refers to the decoupling of directly accessible memory from compute nodes (e.g. servers in a data center rack). Specifically, it entails hardware-enabled system integration which allows compute nodes to directly access memory from other – remote – compute nodes and therefore increase effective memory volume. ThymesisFlow essentially allows servers to access memory from adjacent servers through a custom network as if it was their own and consequently bypass local memory volume restrictions.

Application

Because big data workloads are often hindered in performance by available memory volume and expanding memory volume in data center servers is generally non-linearly expensive [3], traditionally, a scale-out approach is used to scale data center applications for larger data volumes. In a scale-out approach, vast amounts of data are sent over the local network and copied to local memory (Figure 1a), contending for network bandwidth and possibly harming performance by thrashing memory across the compute nodes [3]. ThymesisFlow could potentially mitigate these issues by providing direct access to large memory volumes over a custom network, relaxing the requirement to utilize precious local network bandwidth and the burden to evict data from memory in order to copy (Figure 1b).



(a) Scale-out (b) Memory Disaggregated

Figure 1: Distributed Computation Scaling Approaches

Moreover, by enabling separate compute nodes to access and modify both local and disaggregated data concurrently, without duplicating data, it has the potential to improve application performance. An example use-case would be when compute nodes operate on local in-memory data, while utilizing in-memory data from the other nodes in the network (i.e. wide-dependency operations). This increases the ability of data center workloads to process data in a parallel manner.

Additionally, memory disaggregation could aid in data center utilization strategies by allocating resources dynamically and online across server racks. This allows increased flexibility in server virtualization at high performance and could therefore help data center utilization rates by reducing resource down-time.

Essentially, by efficiently pooling processing and memory resources, the technology has the potential to increase data center utilization rates, decrease total cost of data center ownership, and improve both performance and cost of development for applications [2]. Consequently, it can contribute to improving the efficiency of data center workloads generally and big data workloads in particular.

ThymesisFlow offers a first step towards memory disaggregation in a way that is largely transparent to applications and is already being researched by academia and industry [4]. It is supposed to show a functional proof-of-concept for a scalable and financially viable memory disaggregation system which is integrated in next generation data center processors. IBM has announced memory disaggregation of this kind to be integrated in the upcoming IBM POWER10 processors as 'Memory Inception' [5].

Performance Evaluation

Considering the technology has not yet been rigorously tested with different workloads, there is an opportunity to test its potential to accelerate data center workloads in general. In order to do this, it is important to accumulate accurate data on latency characteristics, for which microbenchmarks of simple data fetching could be used. Ideally, microbenchmarking response latencies in real-world applications, similar to what is already described in [2], would be conducted with more different types of workloads. This data could guide the design of future applications and even server-scale and rack-scale hardware designs.

Furthermore, developing and benchmarking applications that leverage the ThymesisFlow framework against existing solutions provides insights into its general potential to enhance performance. Preferably, existing solutions would be retrofitted with ThymesisFlow support and enhance performance that way. If big data tools and frameworks can leverage the technology successfully, then by extension applications could be able to utilize it as well with minimal to no modification. Alternatively, workloads could be scaled and distributed more easily due to the simplified memory sharing interface. Even if leveraging memory disaggregation in this way yields only comparable performance, the reduced application development cost may be significant.

1.2 Plasma Object Store

Big data applications often consume data from an external source and acquire this data by querying the source. A single source may have multiple consumers querying it. The Apache Arrow framework [6] aims to standardize this supplier-consumer dynamic in an efficient way. Part of this framework is the Plasma in-memory object store, which is used to store and access data within a system in which multiple data suppliers and consumers may exist.

The object store lives as a separate process to which clients of the store may commit and 'seal' data objects with an object identifier. The store manages the objects' locations in shared memory and makes them available to other clients upon sealing. Sealing an object prompts the store to make it immutable, such that sharing it with multiple processes does not yield race conditions. Big data applications often do not require mutability of the source data, e.g. the Resilient Distributed Dataset (RDD) of Apache Spark [7] is also built on this premise.

Plasma store clients can access the existing sealed objects by querying the store for their identifiers. The store then provides the client with a read-only buffer pointing to the object, the client can consecutively retrieve the object data from the buffer. Sharing through system memory ensures that both object commitment and access do not incur large latency penalties. Moreover, the standardized format of the store reduces the need for serialization and deserialization between processes which generally improves performance and efficiency. The framework is already being leveraged in certain big data workloads [8].

1.3 Memory Disaggregated Object Store

A significant limitation of Plasma is the fact that it only supports local object storage. This means that availability of both memory volume and processing units are limited. Memory disaggregation provides an opportunity to create a Plasma implementation that has access to significantly larger volumes of memory. Simply modifying Plasma to allocate in disaggregated memory would already allow applications to use significantly larger data sets without resorting to slow storage devices, scale-out approaches, or stressing the network.

By extension, a closer integration of memory disaggregation would allow multiple compute nodes to operate on a 'pool' of memory. The different compute nodes would then be able to generate and process data from this large pool concurrently. This means that massively parallel tasks could benefit tremendously from the large amount of distributed compute nodes. Especially wide-dependency operations (e.g. shuffle operations) would likely benefit from this type of integration.

Another advantage of the integration of these two frameworks exists. Namely, the fact that Plasma clients can only access immutable data objects. The immutability means that the previously described cache coherency concerns that ThymesisFlow faces do not represent an problem in this case. Since the data objects are placed in disaggregated memory and remain unmodified, they can be accessed by clients without the risk of race conditions.

As it stands currently, memory disaggregation has been featured in research to only limited extent, with the main focus being on general feasibility [3]. Existing technologies have so far demonstrated varying results on smaller systems. At the time of writing, the memory disaggregated big data analysis field is still in an exploratory phase of white papers and feasibility studies, but seems to be gaining traction in research [3][9][10]. ThymesisFlow – and eventually Memory Inception – is the first memory disaggregation solution that is designed for large-scale deployment and utilization. Additionally, it is supported by IBM as a vertically integrated data center technology supplier. It thus has tremendous potential to elicit impactful paradigm shifts in the data center industry.

The current project is set out to perform and benchmark this integration of ThymesisFlow memory disaggregation in the Plasma framework. This paper contains the following main contributions:

- The proposal and implementation of a memory disaggregated Plasma framework, which enables easy and efficient production and consumption of data objects across distributed compute nodes.
- A set of microbenchmarks for measuring latency and throughput of creating and retrieving data objects from the Plasma object store [11].
- Considerations and recommendations for future work on the newly proposed Plasma framework and memory disaggregation technology.

The paper explores application and research of the performance enhancing potential of ThymesisFlow memory disaggregation for big data analytics and provides a stepping stone for potential future work.

The rest of the paper is organized as follows. First, Section 2 provides additional background information on ThymesisFlow. Then, Section 3 presents the new memory disaggregated Plasma framework and corresponding set of microbenchmarks. Section 4 discusses the experimental setup and results. Lastly, Sections 5 and 6 conclude and outline recommendations for future work, while Section 7 elaborates on the paper's research ethics and reproducibility.

2 Background

ThymesisFlow is initially developed for POWER9 [11] architectures and leverages the OpenCAPI [12] interface. The system builds on FPGA accelerators that are interfacing between the ThymesisFlow network and the Linux operating system kernel. The disaggregated memory is exposed to the operating system as memory mapped I/O (MMIO), which is accessible through the ThymesisFlow system such that it becomes transparent to the user [2]. The MMIO mapping ensures that caching is handled gracefully.

Figure 2 shows a schematic representation of the effective physical flow of data in ThymesisFlow. Memory sharing happens through the FPGA accelerators, leveraging the OpenCAPI FPGA stack. OpenCAPI [12], is an interface architecture that enables accelerators to cache coherently access system memory. This means that the FPGA accelerator can access memory from the host in a cache coherent way.

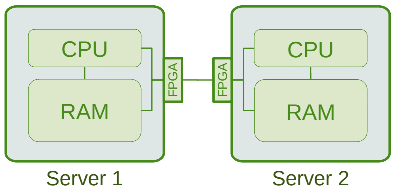


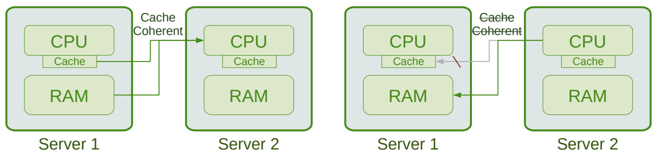
Figure 2: ThymesisFlow Schematic Representation

For ThymesisFlow, this entails that a portion of local system memory is marked as disaggregated and made available to remote compute nodes. The system's FPGA accelerator then represents a kind of MMIO memory controller for remote disaggregated memory. The operating system relays calls for the MMIO to the FPGA, translating memory addresses and requesting the appropriate memory regions from remote compute nodes. The remote compute node's FPGA then uses the OpenCAPI interface to retrieve cache coherent data from the desired memory regions and returns this data to the requesting node. This completes the call to remote disaggregated memory.

As it stands currently, memory disaggregation with ThymesisFlow is subject to several valid concerns as well. For example, calls to disaggregated memory carry an inherent latency penalty due to the extra distance data has to travel on wire relative to local data. This penalty has been observed to be non-negligible [2], thus local memory remains of importance for performant applications [3]. This extra wire distance has to be traversed in scale-out approaches as well and, with efficient custom hardware and network protocols, memory disaggregation could even reduce this latency penalty with respect to traditional local networking.

Moreover, compute nodes using ThymesisFlow could potentially suffer from increased cache coherency and synchronization issues related to the distributed nature of disaggregated memory fetching. When multiple processes are accessing and modifying the same memory region, the cached changes need to be flushed to memory before becoming available to other processes. Usually, cache coherency among shared-memory processes is handled by the operating system kernel, however, with memory disaggregation multiple operating systems are involved. Cache coherency in this setting is not supported by common operating systems and eliminating caching completely – which requires development of custom kernel modules – comes at a cost. Additionally, the increased latency of disaggregated memory calls begs extra caution with race conditions.

The cache coherency concerns arise from the data flow within ThymesisFlow and have important implications for its usage. The OpenCAPI [12] interface ensures that reading remote disaggregated memory is cache coherent (Figure 3a). Alternatively, writing to remote disaggregated memory is cache coherent with the local system, but not necessarily with the remote system. Explicitly, this means that data written to remote disaggregated memory is not necessarily immediately available to the remote system. The written data will be flushed to the remote disaggregated memory, however, the remote system may have cached a previous value (Figure 3b). This has implications for the memory disaggregated Plasma store, which will be discussed in the next section. Note that Figure 3 is only a conceptual schematic representation.



(a) Remote Read

(b) Remote Write

Figure 3: Cache Coherency in ThymesisFlow Transactions

3 Memory Disaggregated Plasma Framework

The goal of the current project is to develop and test a variant of the Apache Arrow Plasma object store [6] that leverages memory disaggregation backed up by ThymesisFlow [2]. This process was conducted in three distinct stages:

1. Setup of a memory disaggregation testing environment
2. Integration of ThymesisFlow into the original Plasma framework
3. Benchmarking of the new Plasma framework

The main yield of the project is the second stage with the third stage providing a context to the framework performance potential, therefore, this paper predominantly focuses on the second stage. This section elaborates on the 3 project stages.

3.1 Testing Environment

Initially, a testing environment had to be setup for development. This was needed due to the fact that no functional ThymesisFlow prototype was directly available during the project. A single POWER9 system – modified to be able to incorporate ThymesisFlow using a loopback design – was available for testing, however, technical issues were encountered when attempting to setup ThymesisFlow on this system. Consequently, a testing environment was created using virtual machines to simulate ThymesisFlow behaviour.

The testing setup was simulated by 2 Linux virtual machines (VMs) run on QEMU-KVM version 4.2.1 [13]. The VMs were both connected to a QEMU inter-VM shared memory object. For this, a memory backend file was created and connected to both VMs as a PCI device. The PCI device can then be memory mapped from inside the VM to use the shared memory similar to how disaggregated memory would function. The final system was tested extensively with this setup.

As discussed in the previous section, ThymesisFlow memory disaggregation differs from traditional shared memory in its cache coherency characteristics. With the inter-VM shared memory functionality [14], Qemu ensures cache coherency among operating systems. Thus, the virtualized test setup clearly differs from a ThymesisFlow system in terms of cache coherency behaviour. This has to be accounted for during testing for the system to remain functional upon deployment on an actual ThymesisFlow prototype.

3.2 ThymesisFlow Plasma Store Integration

The integration of ThymesisFlow in Plasma can be further subdivided in two separate steps:

1. Disaggregated memory allocation; the Plasma store needs to allocate objects in disaggregated memory such that they can be accessed by remote clients.
2. Remote object sharing; the objects contained in the Plasma store need to be shared so that they can be retrieved by clients on all compute nodes.

Remote object sharing is necessary because of the architecture of Plasma; since Plasma clients can only connect with a local Plasma store, they are limited to requesting object buffers from this local store. The local store is not by default aware of remote objects, which gives rise to the need for sharing of remote objects.

For development, the Apache Arrow repository (containing Plasma) [6] was cloned at version 4.0.0. Plasma was originally developed by Ray [15] and is recently adopted by the Apache Arrow repository. Hence, the Apache Arrow implementation was used as a base.

Disaggregated Memory Allocation
As an initial step, the Plasma store was modified to allocate objects in local disaggregated memory. Since the Plasma store is essentially a memory region bookkeeping service for Plasma data objects, it requires sufficiently performant memory allocation. Originally, Plasma uses the infamous Doug Lea's Malloc library (dlmalloc) [16] for this purpose together with a file descriptor system to coordinate memory mapping across Plasma store and clients. This ensures portability, but does not suit the purpose of allocating in disaggregated memory.

Since ThymesisFlow is inherently Linux-based, the loss of portability by substitution of dlmalloc is not a limiting factor. Thus, dlmalloc was replaced by a simpler allocation algorithm that receives the memory mapped local disaggregated memory region and uses it to allocate Plasma objects. The algorithm simply allocates a chunk of memory to the first available region that can accommodate it. By using an ordered map data structure with logarithmic time look-up to keep track of the sizes of available regions, allocation should remain relatively fast. The replacement allocator does not consider e.g. locality, alignment, and fragmentation in memory allocation and thus surrenders some benefits to the original dlmalloc library [16]. It should, however, suffice for the purpose of exploring the performance of memory disaggregated systems.

Remote Object Sharing
Plasma conducts Inter-Process Communication (IPC) between Plasma store and clients through Unix domain sockets. This means that Plasma clients cannot directly communicate with remote Plasma stores as the latter exist in a different operating system, unreachable by the Unix domain sockets. An additional infrastructure could be created to accommodate this, however, that would require all clients to connect with remote Plasma stores and would cause large amounts of duplicate data to be sent over the network. Therefore, it is more convenient to interconnect Plasma stores directly. Additionally, this means the distributed nature can remain hidden to Plasma clients for a large part.

Sharing remote Plasma objects between stores introduces several additional constraints to the system. Obviously, Plasma stores must be able to communicate with each other about currently existing objects. Two new constraints were identified with significant consequences:

- Identifier uniqueness; object identifiers must be unique across the system of all connected Plasma stores.
- Distributed object-usage sharing; Plasma stores should have up-to-date information about which of their local objects are in use by clients system-wide.

The requirement for identifier uniqueness is an immediate consequence of the distributed nature of the proposed framework. If object identifiers are not unique across all existing

Plasma stores, then clients will not be able to retrieve all available objects unless extra logic is introduced for this scenario.

The distributed object-usage sharing constraint relates to a Plasma store's internal policy about evicting objects when needed. Locally, the Plasma store keeps track of which objects are in use by its connected clients. In-use objects will not be evicted, because clients might still be reading from memory and evicting the objects would likely corrupt their data. This logic should be extended to adopt this functionality across several Plasma stores. In the scope of the current project this constraint was considered, however, no solution is currently implemented yet.

A rough subdivision can be made in the possible approaches used to share Plasma object information across the system:

- A shared data structure in disaggregated memory; Plasma stores can share their data structure which maps object identifiers to their corresponding buffers.
- Messaging via disaggregated memory; Plasma stores can perform point-to-point messaging between each other.
- Sharing via LAN; Plasma stores can communicate over the local network using common networking techniques.

The type of approach has far-reaching implications for the system architecture. These implications will be discussed next.

The first approach would allow remote Plasma stores to efficiently look up whether an object already exists and find its corresponding buffer. This approach requires handling issues specific to usage of data structures in shared memory by preventing (local) heap allocation. Moreover, since this is a one-way (local Plasma store to remote store) communication system, there is no way for the remote store to feedback information on currently used objects for the eviction policy. The latter relates to the previously discussed fact that writing to remote disaggregated memory may lead to cache coherency issues on the remote compute node and doing so regardless would open the door to unfavourable race conditions. This could be accommodated by designing a kernel module to disable the memory caching behaviour, but this is outside the scope of the current project.

For the second approach, a messaging system could be implemented, similar to the system suggested in [4]. Messaging in traditional shared memory is a trivial task, however, the cache coherency characteristics complicate the task significantly. This would require developing a robust messaging system using both local and remote disaggregated memory. Any potential performance gain relative to using existing LAN techniques would be marginal considering communication protocol costs and incurs significant additional development cost. A hybrid system which combines disaggregated memory hash map look-up with messaging could perhaps yield more favourable results, but this is outside the scope of the current project.

Lastly, the third approach could be implemented in several ways as well. A simple, robust, and performant approach to do this is based on the Remote Procedure Call (RPC) concept. In this concept, an application can call a function through an RPC client as if it was executed by a remote application. It is a way to hide networking complexity from the application. An efficient implementation for HPC is gRPC [17], which is a high-performance RPC based on Protocol Buffers and HTTP/2 [18]. Internally, a gRPC client connects a stub to a remote gRPC server and relays the local function call to the server, which executes the call and returns the result [17]. This RPC functionality could be used to satisfy the constraints outlined before (Figure 4).

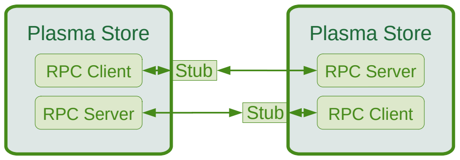


Figure 4: gRPC Functionality

Considering the previously discussed drawbacks of disaggregated memory communication and the simplicity and performance potential of LAN communication, gRPC (version 1.38.0) was used to share objects between Plasma stores. The incorporation of gRPC has a major implication for the remainder of the system.

Due to the requirement of low-latency access to data by Plasma, gRPC is configured in a synchronous manner. Consequently, the gRPC server needs to be available continuously for requests and thus requires a dedicated thread. gRPC clients do not have this requirement as they do not receive requests. Upon a client request for a remote object, the local Plasma store makes an RPC call to look up the object identifier(s) in the remote store hash map and receive the corresponding object buffer(s). Similarly, on object creation, RPC calls are used to ensure uniqueness of object identifiers. This multithreaded look-up introduces the need for thread-safety mechanisms as both the Plasma store main thread and gRPC server thread may attempt to access the hash map structure concurrently.

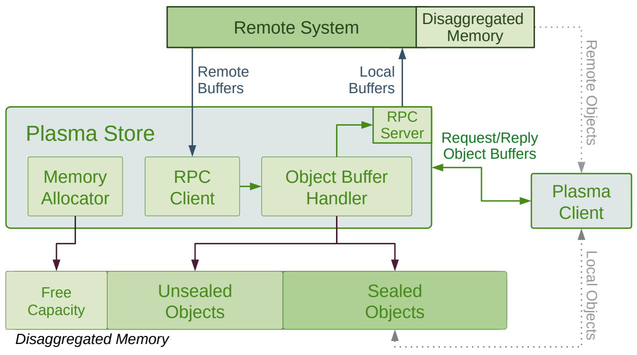


Figure 5: Memory Disaggregated Plasma Framework

To ensure thread-safety, mutex functionality of the hash map in question was built in. This eliminates race conditions among the main Plasma store thread and the gRPC server thread. Since mutex usage can incur significant synchronization costs, it should be kept to a minimum. Since the Plasma store main thread will both read and write to its hash map, while the gRPC server thread will exclusively read from it, adding mutexes to reads from the main thread is not necessary. This arises from the fact that race conditions only occur when either of the threads is writing or will write to the memory region. Hence, only writing from the main thread and reading from the gRPC server thread needs to be protected.

Moreover, gRPC allows to configure single- or bidirectional data streams for large or continuous data requirements. Considering that only data regarding the identifiers and memory regions of Plasma objects needs to be sent, there is likely no performance benefit – perhaps even a performance hit due to additional package overhead – to using streams. Thus, gRPC was configured with unary RPC.

A schematic block diagram with full the proposed system can be found in Figure 5.

3.3 Benchmarking

The last stage of the project was dedicated to benchmarking the full memory disaggregated Plasma system. By benchmarking the system, the premises on which it is built can be verified and its performance can be put in context. Two types of benchmarks can be performed in this case. Firstly, the latency and throughput of retrieving remote Plasma objects can be measured and compared against the same measurements when retrieving local objects. Secondly, the performance of a fully integrated system could be tested against established distributed computing solutions. While the latter could provide valuable information on performance potential in real workloads, it falls outside of the scope of the project.

The current project includes a set of microbenchmarks to quantify the difference in latency and throughput between local and remote Plasma object retrieval. The benchmarks are designed to test a system of at least 2 Plasma stores. Plasma objects with random data are committed to one of the stores and both local and remote clients will then request these objects' buffers from their local Plasma stores and retrieve their data. The data contents should have no influence on the system performance. Gathering buffers and reading the buffers is measured separately. Additionally, measurements are made while creating and writing to the objects to retrieve a reference value. The benchmarks should yield information about the premises on which this system is built and could guide the design of future systems leveraging memory disaggregation and Plasma as well.

6 different benchmarks were included to investigate performance relations in different scenarios, each repeated 100 times to capture the effect of jitter in the system. The benchmarks test Plasma store with different orders of magnitude in object sizes and also vary the number of objects created. This way, the benchmarks can capture differences in local and remote memory performance and variability of full-system performance with different object sizes. The number of objects is varied to mitigate potential influence of caching of smaller objects. The specifications for each benchmark can be found in Table 1.

	Number of Objects	Object Size (kB)
1	1000	1
2	500	10
3	200	100
4	100	1000
5	50	10000
6	10	100000

Table 1: Benchmark Specifications

4 Benchmarking Setup

The main result of this paper is the memory disaggregated Plasma prototype. In order to put the potential of this system prototype in context and to aid the design of future memory disaggregated systems, a benchmarking experiment was attempted as well. This was based on the previously described Plasma system and accompanying microbenchmarks.

The benchmarking experiment was attempted on an IBM Power System AC922 in combination with an Alpha Data ADM-PCIE-9V3 FPGA. The experiment was set up on a single compute node for lack of access to additional nodes. In order to accommodate the single-node setup, a modified ThymesisFlow system with data loopback was used.

Generally, ThymesisFlow uses the Xilinx Aurora protocol and corresponding bitstream IP for link layer network communications [2]. In the applied loopback-modification, this Aurora IP is bypassed such that the transmitting side of ThymesisFlow is directly connected to the receiving side and vice versa. This way, a single node can internally simulate real traffic from ThymesisFlow disaggregated memory calls with similar latency and throughput characteristics as a two-node system. Since the data in the loopback design bypasses part of the networking stack and wire distance, the subsequent latency will be slightly more favourable than in an actual multi-node system. However, considering the small amount of operations and distance, this difference should be minimal. Moreover, an SoC integrated design such as Memory Inception in POWER10 is hypothesized to perform at significantly lower latencies in general [2].

Due to technical issues with ThymesisFlow, only a single functional system existed world-wide at the time of writing. This system was limitedly accessible for experimentation and we were unable to conduct the experiment successfully and collect data within the limited timespan of the project due to system incompatibilities.

5 Discussion & Future Work

Within the limited scope of this paper, the current project explores a new paradigm in distributed computing and its uses for big data analysis, namely memory disaggregation. This new paradigm of memory management in distributed systems requires a different perspective of data management within analysis systems and could transform common data center practices as they stand today. As such, the current paper aims to pioneer the field of memory disaggregation with ThymesisFlow [2] and provide a stepping stone for further research.

5.1 System Demonstration

Unfortunately, we were unable to conduct the outlined experiment within the timespan of the project. This was due to technical issues with the test setup and did not relate to the proposed framework or memory disaggregation technology. At the time of writing, work is ongoing to perform the outlined experiment and evaluate the system performance.

As discussed as part of the experimental setup, a modified ThymesisFlow loopback design was used for testing purposes. This affects the latency and performance characteristics resulting from the experiment relative to a multi-node setup. However, the bypassed Aurora networking IP used in ThymesisFlow, which runs on 401MHz clocks [2] should incur overheads around 55 clock cycles in a non-pipelined approach [19]. Considering the majority of this overhead occurs in the protocol engine [19], this number should quickly diminish in the pipelined design [2]. Moreover, the wire latency should be minimal since rack-scale wire lengths are typically very short.

Furthermore, it should be noted that the full potential of the presented framework can only be tested with solutions that fully integrate it. As motivated before, memory disaggregation introduces a new paradigm in memory management, which can only be fully exploited in specialized solutions. For example, systems relying on wide-dependency operations (e.g. shuffles) will likely benefit most from this technology. This is due to the fact that wide-dependency operations by definition depend on a large volume of remote data in a distributed dataset.

With the presented Plasma framework, compute nodes can perform operations on their local data while retrieving data from remote disaggregated memory, without risking memory thrashing such as in a scale-out approach. In this way, systems integrating the Plasma framework can optimally leverage memory disaggregation without being restricted by cache coherency concerns. More work is needed to evaluate this hypothesis.

5.2 System Improvements

Nevertheless, the newly presented memory disaggregated Plasma system still leaves room for improvement both in terms of functionality and performance. Most of these opportunities have already been touched upon in Section 3 as part of the motivation of design choices. The following paragraphs will elaborate on those opportunities and further future work for the presented Plasma system.

As already discussed briefly, the initial memory allocator was replaced for a simpler and less performant alternative. This was done to enable allocation in local disaggregated memory by the Plasma store and was deemed sufficient for demonstrative purposes. Allocator performance can differ quite substantially [20], so ideally a more performant allocator should be used. The original dlmalloc library is tried and tested as default allocator in several Linux versions [16]. This allocator was originally integrated in Plasma and should provide good performance and configurability when modified to accommodate disaggregated memory allocation, but other viable alternatives exist.

Moreover, the performance of remote object sharing could potentially be improved with an elaborate solution leveraging disaggregated memory. Local Plasma stores could maintain the existing hash map structure for mapping object identifiers to buffer information in their local disaggregated memory. Remote Plasma stores would be able to efficiently look up identifiers in this structure. This direct disaggregated memory look-up likely performs marginally better than the current gRPC solution due to the lack of LAN networking.

If ThymesisFlow cache coherency issues would be resolved by disabling memory caching through a custom kernel module, then remote stores would be able to feedback information regarding e.g. the object eviction policy as well. As discussed in Section 3, tracking which objects are in use by clients shapes the eviction policy, but this is not currently maintained across remote Plasma clients. With a custom kernel module, this could be achieved, however, disabling memory caching could potentially degrade application performance as well. Alternatively, extra RPC functionality could be added to the current solution to attain the same result.

In addition to the current RPC solution, which performs a remote call on every client request for object identifiers that are not available locally, a caching mechanism could be implemented. This could increase performance for repeated requests for identifiers, but the actual effect depends on system usage. Besides, this caching would require caution with tracking object-usage by remote clients for the eviction policy and could result in corrupted object buffers if not handled carefully.

Finally, the currently presented system is able to accommodate a 2 node system. For rack-scale solutions, this needs to be modified to accommodate multiple nodes. The modification should be minor for the current system as most of the implemented functionality can trivially be generalized to more nodes.

Lastly, the currently presented Plasma system was designed for ThymesisFlow, however, it is hypothesized that an SoC integrated solution – such as the announced Memory Inception feature in POWER10 [5] – would carry significant performance improvements [2]. The presented system is modularly designed such that it should be able to use Memory Inception directly or with marginal modification. Memory Inception will be the first scale-manufactured memory disaggregation solution and future work should focus on utilizing and testing its performance with the presented Plasma system.

6 Conclusion

As the topic of memory disaggregation in data centers continues to be become more relevant, the demand for software frameworks that leverage the technology and quantify its potential increases. The current project set out to propose and demonstrate a novel type of in-memory object store based on the Apache Arrow Plasma API [6], which leverages the ThymesisFlow memory disaggregation framework [2]. The introduced Plasma framework pioneers a new paradigm in big data analysis for handling very large data volumes in-memory and leveraging distributed computing in an efficient and application-transparent manner.

The current paper demonstrates the potential for memory disaggregated big data frameworks to transform the field of big data analysis. Nevertheless, with the current state of technology, there is tremendous opportunity for technological improvements and future research in different forms. For example, the introduction of closely integrated memory disaggregation technology such as in Memory Inception [5] and further integration into server- and rack-scale system designs. At the time of writing, the system is already in the process of being microbenchmarked according to the outlined experiment. Furthermore, the introduced Plasma framework should be subjected to more extensive testing in fully integrated applications to quantify its true potential. The current project provides a stepping stone for this future work.

7 Responsible Research

No particular risks for conflicting research ethics were identified in the current project. The paper only includes experiments of the system it introduces on a specified test setup, which exclusively uses publicly available components. The discussed loopback modification for ThymesisFlow is not currently publicly available, however, the experiment could be reproduced on a multi-node setup to retrieve similar results, as motivated. The system source code, benchmarks, and install scripts for dependencies are publicly available on https://gitlab.ewi.tudelft.nl/cse3000/2020-2021/rp-group-64/rp-group-64-rabrahamse. Therefore, the results retrieved from the current project are fully reproducible given the right resources.

References

[1] A. S. Andrae and T. Edler, "On global electricity usage of communication technology: trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.

[2] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "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)*, pp. 868–880, IEEE, 2020.

[3] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, "Memory disaggregation: Research problems and opportunities," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1664–1673, IEEE, 2019.

[4] P. Christian and H. P. Hofstee, "Latest trends in memory dissagregation [webinar]," *IBM*, 2021, March 25.

[5] W. J. Starke, B. W. Thompto, J. A. Stuecheli, and J. E. Moreira, "Ibm's power10 processor," *IEEE Micro*, vol. 41, no. 2, pp. 7–14, 2021.

[6] Apache Software Foundation, "Apache arrow." https://github.com/apache/arrow.

[7] Apache Software Foundation, "Apache spark: Lightning-fast unified analytics engine." https://spark.apache.org/.

[8] T. Ahmad, N. Ahmed, J. Peltenburg, and Z. Al-Ars, "Arrowsam: In-memory genomics data processing using apache arrow," in *2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*, pp. 1–6, IEEE, 2020.

[9] 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)*, pp. 649–667, 2017.

[10] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter," in *Proceedings of the Thirteenth EuroSys Conference*, (New York, NY, USA), Association for Computing Machinery, 2018.

[11] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "Ibm power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017.

[12] OpenCAPI Consortium. https://opencapi.org. Accessed: June 8 2021.

[13] QEMU, "Qemu: The fast! processor emulator." https://www.qemu.org/.

[14] QEMU, "Qemu: Inter-vm shared memory device." https://qemu-project.gitlab.io/qemu/system/ivshmem.html.

[15] Ray, "Ray - fast and simple distributed computing." https://ray.io/.

[16] D. Lea, "A memory allocator." http://gee.cs.oswego.edu/dl/html/malloc.html.

[17] Google, "grpc: A high performance, open source universal rpc framework." https://grpc.io/.

[18] R. Biswas, X. Lu, and D. K. Panda, "Designing a microbenchmark suite to evaluate grpc for tensorflow: Early experiences," *arXiv preprint arXiv:1804.01138*, 2018.

[19] Xilinx Inc, *Aurora 64B/66B LogiCORE IP Product Guide*, April 2018. v11.2.

[20] M. Masmano, I. Ripoll, and A. Crespo, "A comparison of memory allocators for real-time applications," in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pp. 68–76, 2006.