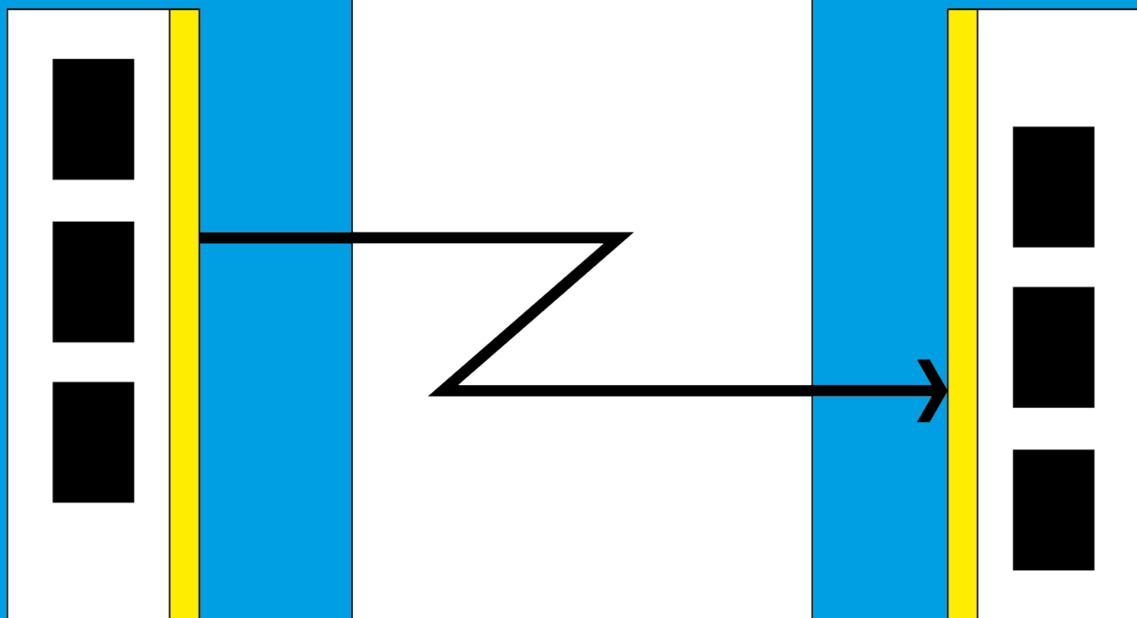
Zero-serialization, Zero-copy memory pooling in compute clusters

Disaggregated memory made accessible

Philip Groet



TU Delft

Zero-serialization, Zero-copy memory pooling in compute clusters

Disaggregated memory made accessible

by

Philip Groet

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday September 20, 2023 at 10:30 AM.

Student number:	4569296	
Project duration:	November 21, 2022 – September 20, 2023	
Thesis committee:	Prof. dr. H. P. Hofstee,	IBM & TU Delft, supervisor
	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. L. Y. Chen,	TU Delft

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

With the rise of the new interconnect standards CXL and previously OpenCAPI, has come a great deal of possibilities to step away from the classical approach where CPUs are in charge of moving data between external devices and local memory. Specifically, OpenCAPI allows for attached devices to directly interface with the host memory bus in a near cache coherent way. IBM has developed the ThymesisFlow [1] system which allows for other servers to access each others Random Access Memory through this OpenCAPI link. ThymesisFlow however is not fully coherent in some cases. ThymesisFlow is designed for the situation where a borrower is able access a lender's memory, and the lender not accessing that borrowed memory. Coherency problems arise in the case where both a lender of memory, as well as a borrower of memory write to the lender's memory. This thesis proposes the use of the *Apache Arrow* in-memory data format to not only access memory in a *near* coherent fashion, but in a *fully* coherent fashion. This will allow compute clusters to more efficiently use memory resources, allow for applications to dynamically hotplug memory, and allow for data sharing without copying over ethernet connection.

The protocols devised in this thesis are able to create disaggregated Arrow objects, which are readable by all nodes in a cluster in a coherent fashion. The creation of these coherent disaggregated objects is the only performance penalty in making them coherent, after initialization all nodes use their local CPU caches to cache remote objects.

A working proof-of-concept has been created which is able to share Apache Arrow objects stored in the memory of a single node. It is also possible to create Arrow objects which span the memory of multiple nodes, allowing for objects bigger than the memory of a single node. The proof-of-concept was able to be run thanks to the setup provided by the Hasso Plattner Institute.

Acknowledgments

This thesis is a collaboration involving many individuals. Numerous people have assisted me along the way, both in technical aspects and on the social front. This thesis would not have come to fruition without the invaluable contributions of these individuals. I would like to express my gratitude.

Peter Hofstee as a supervisor. Your experience in the field of computer architecture was a true eye opener.

Zaid Al-Ars as a supervisor. For introducing me to the world of data analytics in your courses before my thesis, and for the insightful discussions about memory disaggregation.

Lukas Wenzel, Andreas Grapentin and Felix Eberhardt as HPI researchers. For providing access to the HPI ThymesisFlow test setup, and the great mentoring in the usage of the system. Your deep technical expertise into the field of processor architecture is top-notch, and I am grateful you have shared some of your knowledge with me.

Joost Hoozemans for having good discussions about memory disaggregation, giving me feedback, and bringing me into contact with Arrow developers.

Christian Pinto and Charles John as contact persons within IBM. For providing insights into how ThymesisFlow works, explaining the internal processes within the system, and collaborating with me on architectural decisions.

Max Engelen for help in applying this technology to real-world applications

Akos Hadnagy for helping me trying to get the local test setup working at the university.

My friends, fellow students and girlfriend for drinking many cups of coffee, and making a lonely thesis journey a little less lonely.

Contents

Abstract i

Acknowledgments ii

1 Introduction 1
1.1 Underutilized cluster memory . . . 1
1.2 The limits of vertical scaling . . . 1
1.3 Towards zero-copy, zero-serialization . . . 2
1.3.1 Classical data transfers . . . 2
1.3.2 Zero-serialization . . . 3
1.3.3 Zero-copy . . . 3
1.4 Apache Arrow and ThymesisFlow: the perfect match . . . 4
1.5 Earlier works . . . 5
1.6 Problem statement . . . 5
1.7 Structure of this thesis . . . 5

2 Background 7
2.1 What is Apache Arrow . . . 7
2.1.1 Buffers . . . 7
2.1.2 Arrays . . . 7
2.1.3 ChunkedArray: non-contiguous Arrays . . . 8
2.1.4 Columnar data: RecordBatches and Tables . . . 9
2.1.5 RecordBatch descriptor . . . 9
2.1.6 Lifecycle of Arrow objects: std::shared_ptr . . . 10
2.2 What are CPU caches, and how do we keep them coherent . . . 11
2.3 What are computer clusters, and how are they networked . . . 11
2.4 What is OpenCAPI . . . 12
2.4.1 OpenCAPI merged into CXL . . . 12
2.5 How Linux implements Power CPU architecture: ppc64le . . . 12
2.6 What is Linux mmap . . . 12
2.7 Memory disaggregation with ThymesisFlow . . . 13
2.7.1 Comparing ThymesisFlow to SMP and NUMA systems . . . 14
2.7.2 Latency of ThymesisFlow compared to local memory . . . 14
2.7.3 Mapping remote memory to local address space . . . 14
2.8 ThymesisFlow Cache coherency (non-)guarantees . . . 14
2.8.1 Lender OpenCAPI mode . . . 15
2.8.2 Borrower OpenCAPI mode . . . 15
2.8.3 How ThymesisFlow interacts with CPU caches . . . 15

3 Methodology 17
3.1 Inter-node communication using gRPC . . . 18
3.1.1 Synchronous RPC . . . 19
3.2 How to transfer Arrow data . . . 19
3.2.1 The Arrow Inter-Process Communication API (IPC) . . . 20
3.2.2 Considerations in modifying Arrow to serialize table descriptors only . . . 21
3.3 Accessing remote memory using ThymesisFlow and mmap . . . 22
3.4 How and where to allocate Arrow objects . . . 23
3.4.1 Cache line alignment of Arrow objects: 64-byte to 128-byte . . . 24
3.4.2 Resulting Arrow allocator flow . . . 24
3.5 From partial hardware coherency, to full coherency employing software coherency . . . 25

3.5.1 Using Immutability of Arrow objects for coherency . . . 25
3.5.2 Making remote and local caches coherent with flushing, without invalidation . . . 25
3.5.3 Prevent instruction re-ordering to move flush before write . . . 26
3.6 Huge Tables: Spanning single table across all nodes. . . . 27
3.6.1 ChunkedArrays spanning multiple nodes . . . 28
3.6.2 Memory translation done natively . . . 28
3.6.3 Instantiation of huge tables . . . 28
3.7 Synchronizing application state across a cluster . . . 28
3.7.1 Why ThymesisFlow shared memory is not used for state synchronization . . . 29
3.8 Allocating Arrow objects on remote nodes . . . 29
3.8.1 Malloc on remote memory . . . 30
3.8.2 Remote Arrow MemoryPool . . . 30
3.9 Cache coherency in remote memory when writing and reading . . . 31
3.9.1 Reading from memory which was written to by its ”owner” . . . 31
3.9.2 Remote writes: flushing local writes to remote memory from local CPU cache to remote memory . . . 31
3.9.3 Remote writes: invalidating remote cpu cache . . . 32
3.9.4 Flushing is only required during initialization of Arrow objects . . . 32
3.10 Lifetime of a remote object . . . 32
3.11 Overview: how an Arrow object is initialized . . . 33

4 Implementation 34
4.1 Mallocs in mapped regions . . . 34
4.2 CMAKE mods . . . 34
4.2.1 cmake config for compiling protobuf definitions . . . 35
4.3 gRPC . . . 35
4.4 Local MemoryPool and proxied remote MemoryPool . . . 36
4.5 Serializer . . . 36
4.6 Synchronization bit waits . . . 37
4.7 HugeTable logic . . . 37
4.8 Type of flush instructions . . . 38
4.9 Memory barrier instructions . . . 38
4.10 Deletion of objects . . . 38
4.11 Orchestrator: User facing API . . . 39
4.12 Security . . . 39
4.13 Test setup . . . 40
4.13.1 Shared memory mappings . . . 40
4.13.2 QEMU: Shared virtual PCIe memory device . . . 40
4.13.3 ThymesisFlow setup by Hasso Plattner Institute . . . 41

5 Results 42
5.1 Test setup . . . 42
5.1.1 NUMA domains . . . 43
5.2 Linux perf tools to guarantee benchmarks are memory limited . . . 43
5.3 Removing compute bottlenecks in benchmarks . . . 44
5.4 Initializing an Arrow object . . . 45
5.4.1 gRPC overhead . . . 45
5.4.2 Flushing overhead . . . 45
5.4.3 (De-)Serializing table descriptor . . . 46
5.4.4 Malloc . . . 46
5.5 Comparing to a full ethernet copy . . . 46
5.6 ThymesisFlow micro-benchmarks . . . 46
5.6.1 Varying data types and thread count . . . 47
5.6.2 Strided access patterns . . . 48
5.7 Concluding results . . . 49

6 Recommendations and Future work 51
6.1 Instead of flushing, use write-through pages for writing to remote memory . . . 51

6.2 Method of Arrow allocations 51
6.3 More supported data types in table descriptor serializer 52
6.4 Integrate with Arrow Acero execution graphs 52
6.5 Distributed Query languages 52

7 Conclusion **53**

References **55**

A CPython list data structure **57**

B Bring-up of HPI ThymesisFlow setup of Hasso Plattner Institute **58**

1 Introduction

1.1. Underutilized cluster memory

Individual nodes in a cluster are usually provisioned with enough memory for any and all applications it will see in its lifetime. But workloads and virtual machine instances need varying amounts of CPU cores and memory, making it difficult to perfectly match that to the hard limits of the hardware. In addition, users sending jobs to a cluster often over-estimate their memory requirements to prevent their jobs from being inadvertently killed because they ran out of memory.[2] This causes memory in such clusters to usually be under-utilized. A study by Google into its Borg clusters reveals that only around 40% of memory is used [3], and a study by Meta shows 50% of VMs never touch 50% of their memory [4].

Furthermore, data is often copied to multiple nodes, and multiple copies are stored simultaneously. In the case of a big data pipeline for example, if the next step of the pipeline runs on a new machine, it will first need to copy the data it needs to local memory, after which it will run from local memory. This means the data will be in memory multiple times, at the source and at the destination node. Considering that memory is one of the largest contributors to the total cost of a server, it is clear that using memory disaggregation could prevent duplicate data being stored, and be a major cost saver.

1.2. The limits of vertical scaling

The industry is reaching a bottleneck regarding the performance of a single CPU core. For large workloads multiple cores are attached together in which they all belong to the same ”cache coherence domains”. In an average laptop we have multicore processors, and in an average server we have multiple multicore processors. Cache coherence domains simply means that within a system all memory is accessible by all CPU cores, and all caches in he CPU cores are guaranteed to contain correct data. IBM has developed SMP systems which even bond together the coherency domains of multiple servers together to form a rack scale coherent system.

Scaling these multiprocessor systems is also reaching its limits. Keeping an IBM SMP system coherent requires a very high bandwidth interconnect between servers, and thus incurs a performance bottleneck, which limits the scaling to a single rack of servers. Were we to scale beyond a rack of servers, the solution is usually to employ a networked cluster of servers. Supercomputers such as the currently biggest from the Top500 list, Frontier at OLCF [5], are interconnected with HPE slingshot [6]. HPE Slingshot is built on top of the ethernet protocol in which data is serialized, encapsulated, sent, and finally deserialized into the destination server [7].

When data needs to be interpretable by other machines their format needs to allow for that. In many data structures this is not possible. Serialization is used to convert data into a format which is readable by other machines.

A challenge with networking nodes together with ethernet interconnects is that communication between

nodes is relatively expensive. Every time a data packet needs to be sent it is serialized into a standalone data packet, copied to local ethernet buffers, transmitted, stored on the receiving side, and finally deserialized on the receiving side. Some systems will have optimized some of these steps away, such as having the ethernet card directly write into memory. These serialization steps are very expensive and require an application developer to carefully design a communication system for a cluster.

This brings the current paradigms of scaling to one of two topologies:

- Spreading the problem space to multiple nodes. Multiple nodes are given a distinct task to execute, and data is either fully stored on the server, or data is broadcasted to every node. Fully storing copies of the data on every node is very inefficient as previously discussed. Broadcasting the data to every node when needed is very expensive as it has high latency
- Symmetric MultiProcessing systems. These systems have a high speed interconnect between them to couple CPUs together. A lot of complexity and bandwidth is spent on getting all these caches coherent. These systems are limited to rack scale computers, as any bigger and the bandwidth needed between servers would be unrealistically high.

Ideally a supercomputer or cluster should have a very low penalty in communicating in-between server nodes. Both the serialization of data, as the copying of data is a major bottleneck in supercomputers which this thesis aims to minimize. A coherent system, with more than a rack of servers, allowing servers to interact with each others memory is the goal of this thesis.

1.3. Towards zero-copy, zero-serialization

In an orchestrated compute cluster we want to transfer data in-between compute nodes. If we want to divide the work between nodes, every node will have to have access to the full data set. This is where the challenges start. Copying the full data set to every node is slow, and consumes a lot of expensive memory. Data may for example be stored in an array of pointers to the records. These records are spread out throughout the programs memory. If we would want to copy this data we would have to resolve any pointers, and pack all the data into a single transmittable piece of data. This conversion of data from a locally readable format, is an expensive serialization step.

1.3.1. Classical data transfers

Ethernet is not a very fast communication protocol, as it has a lot of steps in between where the data is buffered, encapsulated, split into parts and combined again. Take Figure 1.1. When we want to copy data form the left server to the right, we first need to serialize our data into a format understandable by the other server. Then transmit it to the other server through an ethernet-like connection, and finally the other server needs to deserialize the data packet into a performant local representation of the data.

Some High Performance Clusters may employ a faster alternative to ethernet, an often used protocol is Infiniband for example. Infiniband allows for very high bandwidth, low latency, communications compared to ethernet. It is used on the majority of the HPC systems on the Top500 list. [8]

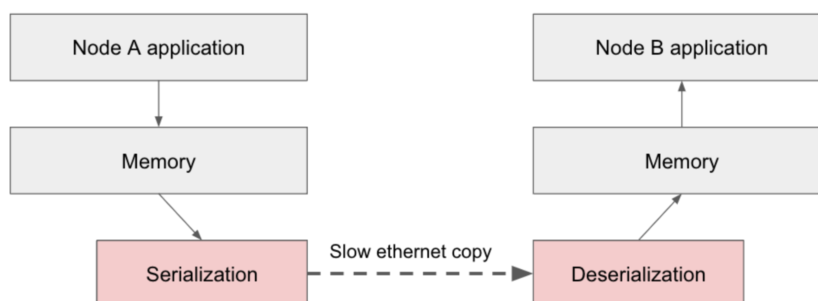


Figure 1.1: Transferring data across two nodes. Both serialization and copying are expensive operations.

1.3.2. Zero-serialization

The most obvious step to improve on the classical approach to data transfers is to have the systems speak a *common language*. If the applications store their data structures in a platform and machine agnostic way, we skip all the serialization steps. This is what Apache Arrow aims to achieve, an in-memory data format which is understandable across applications.

Let's take for example the difference in the way C++ and Python store a simple list. By default C++ allows to only store homogeneous data in a list: every element in the list should have the same type. Python however allows for storing heterogeneous data in lists, and also variable in size. Python allows us to have list which contains variable-length strings, integers and other lists mixed within a single list. This is possible as Python stores its data with an abstraction layer called *PyListObject* (Appendix A). C++ stores arrays in a more compact and optimized way than the CPython backend, and allows the developer to be very flexible wit data types. The downside is that the format is not uniform across data types, not uniform across user applications, and not uniform across CPU architectures. A C++ data object and Python data objects are not at all interchangeable. Python data objects have a very fixed structure which they must follow, which C++ does not understand. While C++ data objects are designed by the application or developer which are not interoperable with Python objects. When we want to trasfer objects between these to languages we would need to serialize, copy, and deserialize.

But also for interchanging data between two C++ programs we have serialization steps. Due to the way virtual address and physical address spaces are constructed in modern operating systems, memory addresses in one application are not valid in the memory space of another application. Every process has its own virtual memory address space which the processor and kernel map to a certain hardware memory address. Because every process has its own unique memory address space pointers cannot be simply sent and understood by another process. Let's take for example a table of strings as depicted in Table 1.1. If we were to copy this object to another process, it will probably be placed at a different address space, causing all pointers to memory addresses to be invalid.

Address	Data
0x7f040	Element 1: 0x7f080 Element 2: 0x7f0c0 Element 3: 0x7f0c2
0x7f080	"abcdefg" 0x00
0x7f0c0	"i" 0x00 "jkl" 0x00

Table 1.1: Example list of arrays in C++, depicted as array of pointers

Arrow unifies the data format so that both Python and C++ store data in the same performant way, but also that processors with differing architecture can still communicate with each other. Every object is made to be portable across processes. Arrow for example requires all data to be little endian by default, and by default it requires all data to be aligned on 64-bit addresses to allow for Intel SIMD architectures to process the data more efficiently. It standardizes complex data structures such as dictionaries and variable width arrays.

Apache Arrow allows applications to speak the same language, and thus ensures that between application we do not need to translate/serialize data for the other application to understand it. This allows us to transfer data by simply copying the whole data structure, and not think about how it will be interpreted by the other side, and prevent the receiving side from having to convert the data. As serialization is a major bottleneck in big data pipelines, this gives a big performance boost. Comparing the system depicted in Figure 1.1 with the Arrow situation in Figure 1.2 we now have that Arrow removes the serialization steps.

1.3.3. Zero-copy

We discussed how to solve the biggest bottleneck using Apache Arrow compared to the classical approach: data serialization. The next bottleneck in the system is copying the actual data. We aim to not only prevent serialization, but also prevent the expensive copy operations. Having data stored in two places is time-expensive, and copying it is expensive.

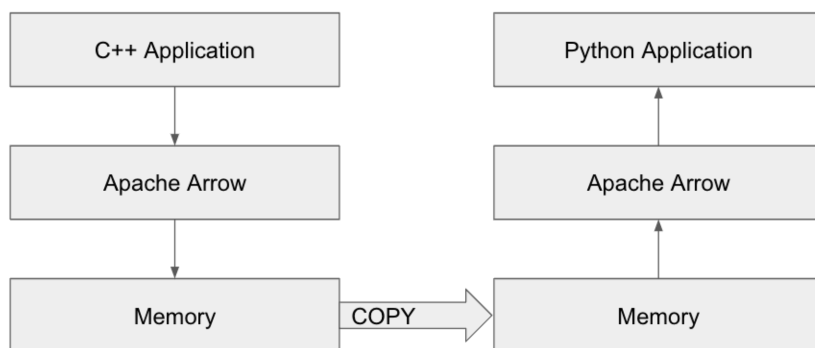


Figure 1.2: No serialization transfer with Apache Arrow

The final step is to remove this copying and duplicate storing of data. A research prototype by IBM called ThymesisFlow [1] is useful for this case. This prototype uses OpenCAPI to interconnect two machines, allowing them to read and write into each others memory in a *near* cache coherent fashion. ThymesisFlow allows a processor to call any memory instruction on a remote address space, and it will be executed on the remote machine, while still using local CPU caches. It even provides a user-friendly API to *mmap* remote memory into application userspace memory. This allows for different servers to access each others data without having to store a local copy of that data. This allows the user application to use any memory instruction of the processor on remote memory as if it is local. Because ThymesisFlow makes use of the processor's architecture for memory management, all CPU caches are used during ld/st instructions. Limitations of ThymesisFlow however are similar to other interconnect standards: low interconnect bandwidth. [9] How ThymesisFlow works, and what OpenCAPI is, is explained in section 2.4.

ThymesisFlow removes the final bottleneck in our data transfer pipeline. Adding ThymesisFlow to Figure 1.2 gives Figure 1.3, where the copy bottleneck has now been removed.

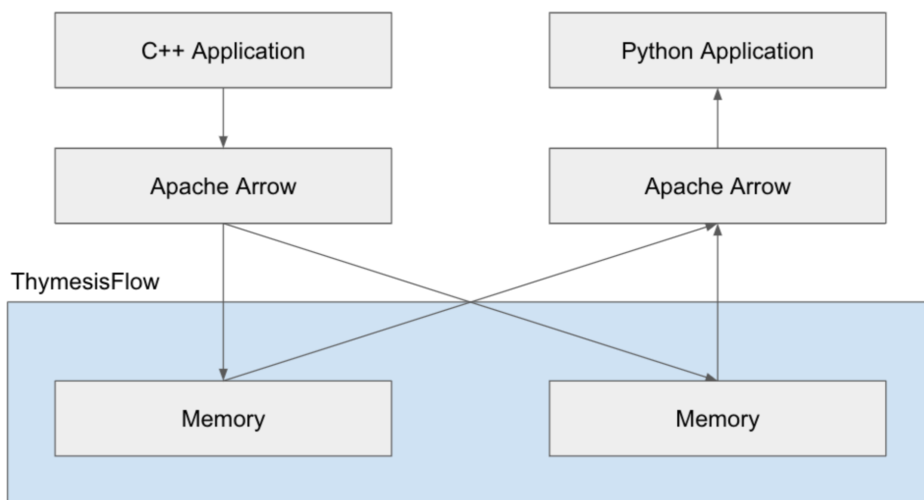


Figure 1.3: No serialization, no copy transfer with Apache Arrow and ThymesisFlow

1.4. Apache Arrow and ThymesisFlow: the perfect match

Ideally we want to be able to read all data of every node in a cluster without having to worry about the data format it is in, or where it is stored. We combine the memory transfer architecture of ThymesisFlow/OpenCAPI with the common data format provided by Arrow.

We use the ThymesisFlow architecture to allow for transparent memory transfers between compute

nodes, in a low-latency, identical address space fashion. If we want to share memory of one memory ”owning” node, called a lender, we simple map the lender memory into the memory of a borrower node. Memory of the lender node is now accessible from both the lender as well as the borrower node as if it is local memory. Any instructions issued which reference a shared region are fully transparently relayed to a remote node.

Arrow then assures that the data format used is fully portable between servers. With the standardized Arrow format, any user application regardless of programming language and system, is able to read the data, and execute operations on that data. All without having to re-interpret, copy or translate the data. We point our application towards a certain address, ThymesisFlow ensures st/ld instructions are executed remotely transparently, Arrow allows us to correctly read and write the typed data.

To summarize, the work in this thesis allows for the following optimizations in compute clusters:

1. Memory in clusters is more optimally used. No longer do servers need to be given the amount of memory it will maximally need. Memory can be hot-plugged when a resource intensive application is run.
2. Data in a cluster no longer needs to be copied across nodes. Expensive transfers, expensive duplicate local copies, and expensive serialization steps are omitted.
3. When data is only sparsely used, it no longer needs to be copied across in its entirety. ThymesisFlow allows for individual data accesses.

1.5. Earlier works

The use case of disaggregated memory to be accessed by both local and remote nodes through the ThymesisFlow system has been accesses before [10]. The authors in this paper have modified the Apache Arrow Plasma store as a baseline for creating disaggregated memory stores. However Arrow Plasma has been deprecated and no longer in active development. Furthermore there are some coherency issues which the paper does not touch upon, and the authors were not able to run their experiments due to technical difficulties with the setup.

Furthermore many disaggregation technologies have only become available after 2005. Before that, the promise of memory disaggregation had been discussed, the implications analyzed, but no hardware implementation was made before that. [11]

1.6. Problem statement

The goals of this thesis can be summarized as follows:

Research Question 1 Can memory disaggregation be made more accessible by combining ThymesisFlow and Apache Arrow?

Research Question 2 What cache coherency issues does ThymesisFlow have when multiple nodes access the same memory region? Can we work around those issues?

Research Question 3 Is Apache Arrow a good fit as a memory disaggregation friendly data format?

1.7. Structure of this thesis

This thesis will start of with explaining the concepts behind Apache Arrow and ThymesisFlow in the Background chapter (chapter 2). A thorough analysis is done on how these systems work, and what technologies lie behind them. As these technologies both need to be modified to work in a memory sharing context, an analysis is given what the shortcomings are for the use case described in this thesis.

The methodology chapter will explain how the located challenges of the background section are solved. It explains on which systematic solutions have been made, and explains how these systems should

work in a high-level sense. An explanation is given on how the nature of Apache Arrow helps in making the system cache coherent.

In the implementation chapter the systems described in the methodology section are implemented into actual code. It gives a translation of the systems described in the methodology sections to the code paths that will be followed and which modifications to Apache Arrow have been made.

Finally in the Results chapter measurements are done on the created systems. The time it takes to initialize a disaggregated object is measured, and some guideline measurements are given to estimate what the performance if the system will be for several use cases.

The Discussion chapter will contain recommendation for future work, and will give some suggestions for how ThymesisFlow and Arrow can be architecturally changed to ease disaggregation applications.

2 Background

2.1. What is Apache Arrow

In this section an explanation is given on what Apache Arrow is, and how it works internally. Any relevant details are explained on which this thesis builds on later.

Apache Arrow is a standardized format for storing in-memory data. Instead of letting programming languages and compilers decide what the data structure of data is, Apache Arrow formalizes it in an application agnostic way. When for example two applications, one Python, and one C, want to talk with each other often an intermediate language is used such as JSON, or something similar. As discussed in the introduction, this intermediate language is cause for a great deal of inefficiencies.

Arrow allows for different user applications to understand the same data. When an application wants to communicate a piece of data using Arrow, it will simply copy the Arrow data into a shared memory region, and the other application is already able to read it. No serialization of the data is needed, the other application is immediately able to read the data.

Arrow not only helps in inter-process communication within a single machine, it also helps in communicating between multiple machines. To copy an Arrow object to another machine we simply copy the Arrow object as is to the other machine using for example an ethernet link. The other side simply stores this received data as is, and is immediately able to read it. The power here again is that no serialization and deserialization steps are needed to make the data transmittable or interpretable.

2.1.1. Buffers

The lowest abstraction layer Arrow provides is a Buffer. A buffer is a single contiguous piece of memory, which can be either mutable or immutable. The buffer contains a memory address, as also the size of the underlying data buffer. A buffer keeps track how big an underlying memory space is, and what its current capacity is. Almost all Apache Arrow objects use Buffers to describe contiguous chunks of memory.

The data format contained inside a buffer is dependent on the abstraction layers above it. For example when we store an array of fixed size integers, the Arrow Buffer inside it will sequentially store the data. But if we store a variable width data type in an array, two Arrow Buffer objects are used. One for the data itself sequentially stored, and an offset buffer which points to the index in the data buffer where every element is stored.

2.1.2. Arrays

Apache Arrow Arrays contain Arrow Buffers with data, the Arrays also describe the structure of the data contained in the Buffer. For example the Array object holds the type information, the number of elements, while the Buffer simply stores the data and keeps track of the amount of bytes stored.

It is up to the application using Arrow to decide how the buffers are created, Arrow handles combining the raw data buffers into a processable Arrow object. To illustrate this let us take the two different ways an Arrow array can be created:

1. Using Array builders we can dynamically add data to an array. At the end of the build cycle we finish the array, which tells arrow to create the Arrow Array object and flag it as immutable.
2. Wrap an existing array in memory inside Arrow Buffer abstractions, which can then be wrapped into an Arrow Array.

The first method allows applications to dynamically build up an Array. Arrow allows for defining the expected size of the array to allow Arrow to pre-allocate memory, instead having to dynamically re-allocate the buffers as the data grows. An application will first create an ArrayBuilder instance of the requested type, e.g. 8-bit integers builder. Next the application iteratively adds values using the AppendValues methods. Finally the Array is *finished*, as Arrow will not allow any more modifications. The Arrow Buffer object is made immutable, is wrapped inside the ArrayData abstraction, and finally the Arrow Array object is returned. Below is this basic example. An important note here is that the data is immutable after it is finalized, this will become clear in section 3.5 why this is important.

```
1 arrow::Int8Builder int8builder;
2 int8_t* days_raw = (int8_t *)malloc(SIZE);
3 for (unsigned long i = 0; i < SIZE; i++) {
4     days_raw[i] = (int8_t)(i*2 & 0xFF);
5 }
6 ARROW_RETURN_NOT_OK(int8builder.AppendValues(days_raw, SIZE));
7 ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::Array> days, int8builder.Finish());
```

It should be noted that this is not the most performant code. Arrow does not know beforehand how big the array is going to be, and has a greedy approach to allocating the memory regions. When an array builder wants to append more values than has been allocated, Arrow will (at least) double the capacity of the array, causing a reallocation event. Reallocation events are very expensive as a new, bigger, region is allocated, all data is copied over to the new region. The code used by Arrow to grow the capacity is given below. Also, data is copied from the *days_raw* buffer into an Arrow allocated buffer. Ideally we would tell Arrow how big the data is going to be using *builder.Reserve(SIZE);*, but then still Arrow will copy the data from the days_raw buffer into an Arrow Buffer.

```
1 static int64_t GrowByFactor(int64_t current_capacity, int64_t new_capacity) {
2     return std::max(new_capacity, current_capacity * 2);
3 }
```

In the second method we manually construct the Arrow abstractions around the data. This is not the recommended route to create arrays as it allows the application to construct invalid schema, or corrupt Arrays. But it is the fastest way. Arrow does allow for validations, partial or full, but these can be computationally expensive. Below an example of how to wrap an existing buffer in in an Arrow::Buffer object, and then wrapping that in ArrayData and finally in an Array object.

```
1 std::shared_ptr<arrow::Buffer> buffer;
2 ARROW_ASSIGN_OR_RAISE(buffer, arrow::AllocateBuffer(1000));
3 int8_t* buffer_data = (int8_t*)buffer->mutable_data();
4 for (unsigned long i = 0; i < buffer->size(); i++) {
5     buffer_data[i] = (int8_t)(i*2 & 0xFF);
6 }
7 auto data = arrow::ArrayData::Make(arrow::int8(), buffer->size(), {nullptr, buffer});
8 std::shared_ptr<arrow::Array> days = MakeArray(data);
9 ARROW_RETURN_NOT_OK(days->ValidateFull()); // Or Validate() for structure validation only
```

2.1.3. ChunkedArray: non-contiguous Arrays

Arrow supports combining multiple arrays into a single *ChunkedArray*. These ChunkedArrays are saved as a list of Arrays. Note that while Array objects are guaranteed to contain a contiguous chunk of memory, ChunkedArrays therefore are itself non-contiguous, but do contain contiguous chunks of memory.

ChunkedArrays allow for a collection of Arrays to form a single indexable array. When indexing the ChunkedArray, Arrow will calculate in which Array that index resides, and return that value.

The ChunkedArray abstraction is not part of the Arrow format specification, rather it is a library abstraction which makes use of the Array objects. When for example the application requests a compute kernel to run over the ChunkedArray instance, the compute kernel must support ChunkedArrays by which it will loop through the Arrays contained within the ChunkedArray.

2.1.4. Columnar data: RecordBatches and Tables

Arranging Arrays into a tabular format gives a *RecordBatch*. RecordBatches are a collection of Arrays where every Array is a column in the RecordBatch. Every column of the RecordBatch should contain the same amount of rows. Important to note here is that RecordBatches may only contain Arrays, they may not contain ChunkedArrays. RecordBatches columns therefore ensure that all columns take on the Array property of containing a contiguous chunk of memory.

When combining ChunkedArrays into columnar data we get a Table. Tables do not have the property of contiguous columns, as ChunkedArrays may contain multiple non-contiguous buffers.

An interesting feature is that RecordBatches can be combined into a single Table. The columns of the RecordBatch will each be concatenated into a ChunkedArray, which forms a table. The other way around is also supported, from a table we can derive RecordBatches. Every RecordBatch extracted will have columns Arrays which are derived from the ChunkedArray of the source Table (Figure 2.1). Because not every column may be chunked the same, Arrow will divide the contiguous pieces of memory into smaller contiguous pieces of memory called *Slices*. From these slices we may then construct equal row length RecordBatches.

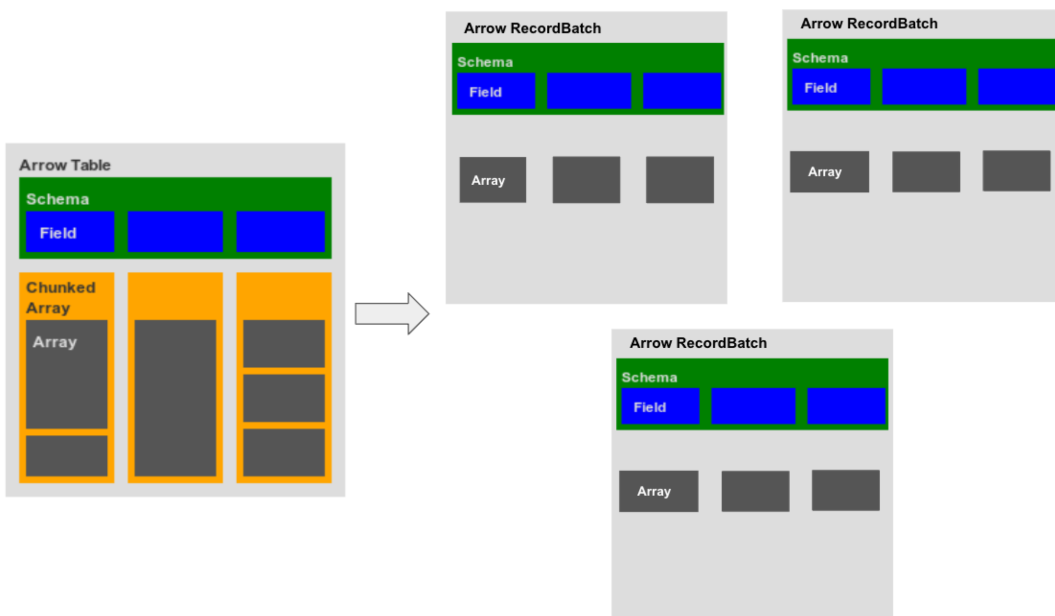


Figure 2.1: Splitting Arrow Tables into RecordBatches

Note that because a RecordBatch contains contiguous columns, it is column oriented. This is unlike for example Apache Spark, where the rows are contiguous memory buffers. Practically what this means is that in Arrow to look up a row we need to do an offset calculation for every column, but if we want to retrieve a column we only need to do one. While in Spark when a column needs to be retrieved, an offset calculation needs to be done for every row, and retrieving a row only requires one.

2.1.5. RecordBatch descriptor

Its important to study how the *structure* of Arrow data is stored, as we want to be able to parse this structure on a remote node. For this thesis we use the word *descriptor* to describe the data types contained, the length of the data, and pointers to the underlying memory data buffers. The word *descriptor*

is not used to describe the data itself, merely the structure and pointers to the data.

Arrow stores some of its internal data in Arrow Buffer objects, while other data is stored in ”compiler allocated storage”. Compiler allocated storage being for example the C++ compiler deciding how it stores class instances or struct instances. This is relevant as data stored in compiler allocated storage is again non-portable. When Arrow transfers for example a table from one application to the other it serializes this structure data, where it is deserialized at the other side. Note that this descriptor is small, and quickly serializable. In the Results section (chapter 5) we will analyze the performance.

Let us take a look into how a RecordBatch is represented in the C++ Arrow library code. A RecordBatch class instance contains a an Arrow Schema and a series of Arrow Arrays stored as a C++ vector as columns. The schema contains the typing of the columns, column names, user metadata, nullability, and any subtypes for nested types.

Arrow Array objects are immutable, and contains a reference to a single ArrayData object. The ArrayData abstraction is a mutable container of an immutable piece of memory. The ArrayData class describes the contents of an Array and may be used to cast one type to the other. Inside the ArrayData we have a Buffer object. The Buffer object contains a pointer to a memory region, and keeps track of its size and capacity.

The way in which Arrow stores descriptor data is not defined by the Arrow specification. this is left up to the application to decide, as it allows for creating abstraction layers. In C++ for example classes are used, where the descriptor data is stored in class member variables. If another application would want to process the Arrow RecordBatch structure, it would need to have to reinstantiate this structure up until the pointer inside the Buffer. The data inside the buffer is defined by the Arrow format specification. It would not need to transfer the contents of the data, only the structure describing the schema, types, and buffer locations. All these abstractions are stored in memory allocated by the compiler.

2.1.6. Lifecycle of Arrow objects: std::shared_ptr

The lifetime of an Arrow object is dependent on the platform Arrow is used on. On C++ for example many of the class instances are wrapped in *shared_ptr* classes. As soon as an Arrow object with shared_ptr goes out of scope, or the last reference to the object was removed, the shared_ptr will delete/free the object from memory. This delete is then propagated down the class inheritance chain untill it comes to the Arrow Buffer object which calls free on the contained memory address. In Python the the Arrow implementation is dependent on the garbage collector to check for any unused memory. In this thesis we will focus on the C++ implementation of the Arrow format.

Arrow data objects are immutable once created. As soon as the data becomes an Arrow object, they are not allowed to be modified anymore. This is enforced on a library level. As every operation on the data should go through the Arrow libraries, data is ensured not to be modified. For Arrow usecases this is usually not a problem. Combining the data format with the Execution Plan Engine, Arrow is able to lazily execute computations, resulting in a completely new Arrow dataset. The idea is that when data is transformed, it is never transformed in-place, but is rather transformed into a new Arrow object. The old object is automatically discarded in the C++ case when all shared_ptr’s go out of scope, or in the Python case when all references have been overwritten or lost.

In this thesis we modify the lifecycle of an Arrow object for several different use cases. For a baseline reference the standard lifecycle of an Arrow object is given as:

1. Creation: Write into memory, special Arrow Buffers are mutable
2. Finalization: Buffers are wrapped in Array, ArrayData, RecordBatch abstractions. Data is made immutable.
3. Usage: Data is able to be read from the buffers
4. Deletion: Dependent on the platform, data is removed form memory. In C++ case this happens when all references of the object have gone out of scope.

2.2. What are CPU caches, and how do we keep them coherent

A computer consists of one or more Central Processing Units (CPUs). Each of these CPUs usually have multiple cores in them. Every core is able to do computations independently of the other cores, and is able to run completely different programs at the same time.

Within the CPU there is not a lot of memory, only small memory regions called registers are available. When an application wants to store larger pieces of data it will have to access the so called *Random Access Memory (RAM)*. RAM is stored on a separate chip from the processor, and is thus relatively slow to access. When a processor frequently accesses a certain address in memory, it is inefficient when it has to re-fetch it every time. Rather, the processor has local caches, so it only has to retrieve data once from memory, after which it will be stored in CPU caches which are on the same chip as the CPU. See Figure 2.2.

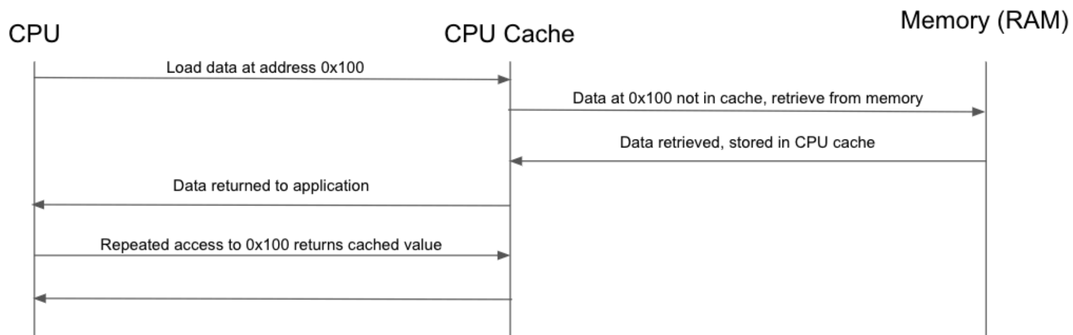


Figure 2.2: How CPU caches are populated. A value fetched from memory is stored in the CPU cache when it is requested later on.

2.3. What are computer clusters, and how are they networked

When a lot of compute power is needed, it is not possible to simply buy a better processor. A machine may be able to have multiple processors, but those are usually limited to 4 sockets for CPUs. An expensive solution would be to employ one of IBMs SMP systems. In these systems multiple servers have their CPUs wired together to allow them to act as a single big computer. These systems can run a single instance of Linux, and programs are able to access memory on all other servers. These systems can also be partitioned into smaller parts each running their own OS. However, SMP systems are also limited in size. SMP systems do not scale beyond a rack of servers, due to bandwidth limits in the interconnects between them. To keep he aforementioned CPU caches coherent with each other, a lot of data needs to be interchanged.

A cheaper and scalable approach is to network multiple servers together using a routed setup. Servers each have a network card which allows them to send routed messages to other servers. Every server will only be able to access its own memory, and every server runs its own Operating System instance. This is different from the SMP setup, no cache line traffic needs to be communicated to the other servers, as everything is local to the server. The downside however is that ethernet connections have quite a high latency and are inefficient in their storing of data. This paradigm usually means that when another server needs data stored on another server, it needs to be copied locally before operations can be run on it.

An especially slow part of these ethernet connected systems is the CPU which is in between the network card and the memory. When data needs to be sent, the CPU of the transmitter is responsible for copying the data from the memory into a memory buffer of the network card. And on the receiving side the same happens, the CPU is responsible for receiving data from the network card and copying it into memory.

2.4. What is OpenCAPI

In previous section 2.3 we studied how CPUs are responsible for moving data around in the computer. A big movement in the computer architecture industry is to remove the CPU out of the loop of every data transfer that occurs. One such technology is OpenCAPI, Open Coherent Accelerator Processor Interface. IBM has developed the original specification, and has donated it into the OpenCAPI consortium. The specification aims to provide a high speed cache coherent interconnect between devices. It allows for connected devices to access memory without the CPU having to be in-between. An OpenCAPI supported network card for example would be able to write directly into the memory of a machine, while maintaining cache coherency for the connected processors.

The cache coherence part is quite a challenge. If for example an OpenCAPI enabled GPU were to read memory, while a change of memory is pending in the CPU cache, the GPU will retrieve the outdated value from memory, instead of the new value pending in the CPU cache. When a GPU requests memory, the OpenCAPI bus will first snoop the CPU cache, if the value is not present in cache it will be retrieved form the actual memory.

OpenCAPI also allows CPUs to access memory inside accelerators in a cache coherent fashion. If for example an OpenCPAI GPU has its own memory banks, the CPU is able to read from those memory banks as if they are local to the CPU. Interesting here is that the CPU can use any memory instruction it knows on the memory address of the accelerator. The OpenCAPI protocol will transparently bridge memory requests to the correct attached device.

2.4.1. OpenCAPI merged into CXL

OpenCAPI is not the only bus interconnect on he market. OpenCAPI was created by IBM, later turned into a consortium where several industry partners partook in. CXL is the interconnect which is originally primarily developed by Intel. CXL has a lot of similarities with openCAPI, and has a lot of the same ideas. In 2022 all assets of the OpenCAPI consortium were transferred into the CXL consortium.

The technology and ideas which this thesis has developed should be transferable to the CXL standard.

2.5. How Linux implements Power CPU architecture: ppc64le

Linux supports the 64-bit PowerPC architecture architecture under the *ppc64le* name. The address space layout differs from x86 in that the POWER PC architecture has a very different memory management system. In Power the address space an application can access is called an Effective Address, or EA for short. [12]

The user application issues instructions with 64-bit effective addresses. The effective address space is divided up into parts with each their own function. These effective addresses are translated by a so called segment table to a *virtual address*. The 64-bit effective addresses are translated by looking up the first 36-bits in a segment register table which converts the 36-bit segment register to a virtual segment ID of 52 bits. This virtual segment ID is prefixed to the remaining effective address bits to form a 80-bit virtual address.[12]

The Virtual Address space is the space as seen by the kernel. This address space addresses over 1000000000000 terabytes of addresses. When the kernel issues memory requests to a virtual address, the processor converts the virtual address into an *Absolute Address*. The absolute address is the address used internally by the processor on the main processor bus. For a regular RAM memory access for example, the virtual address will be translated to an absolute address which lies within the absolute address region to which the RAM is mapped. [13]

2.6. What is Linux mmap

This thesis makes extensive use of the Linux mmap syscall. This syscall allows for mapping memory (devices) into the memory of a process. An example would be to map a file on disk to the userspace memory of a process. Any writes to the mapped memory, will be written to the file by the OS. It is

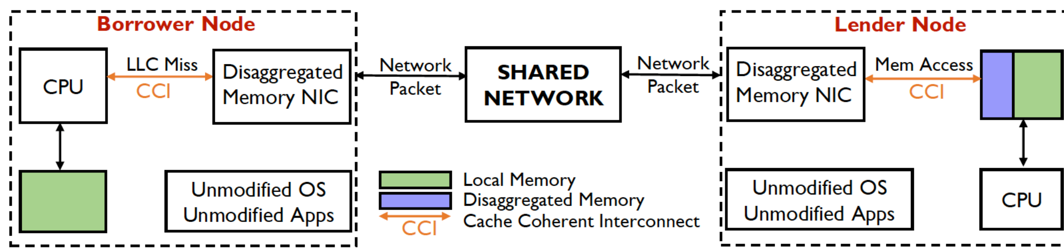


Figure 2.3: Basic architecture of ThymesisFlow memory borrowing system. [16]

also possible to map devices into memory, for example a PCIe RAM stick would be mappable to the application memory, allowing for transparent memory accesses to the PCIe device. An especially useful property is that the location where memory is mapped into the memory of a process, is variable. By default the mmap region is placed at a Linux decided location.

To stay in terms of the Power architecture we will use the terms Effective address for describing process userspace address space, virtual address space for the space the kernel works in, and absolute addresses for the addresses seen on the physical processor bus.

When calling mmap the user application requests a certain memory, or device, to be mapped into the effective address space. After which the application can issue instructions referring to addresses in the mapped region, which will be redirected to the underlying absolute address space. The operating system achieves this by inserting a set of entries into the previously described (section 2.5) segment register tables.

By default mmap decides where the mapped regions is mapped in the effective address space. We can influence the location where mmap places the memory mapping by passing in a ”hint”. mmap will then find the nearest free, page aligned address to map into. We can further restrict the mmap call by passing in the *MAP_FIXED* flag. This flag tells mmap to not take the passed address as a hint, but to take it as an exact location. Using the MAP_FIXED should be done with care though, if the region overlaps with another mapping, the other mapping will be discarded. [14]

2.7. Memory disaggregation with ThymesisFlow

ThymesisFlow is developed by IBM and allows for pooling memory of multiple systems into a single big memory pool. It allows for one server, called the lender, to share (*lend*) its memory to a compute node, called a borrower. See Figure 2.3. The borrower node is able to access the lender memory as if it is local memory from an address point of view. This allows for a compute node to not only use local memory, but also use remote memory. The compute node will then essentially have doubled the amount of memory it can access. Dynamically sharing memory across a compute cluster will ensure memory is more efficiently allocated to the compute nodes which need more memory. Previously, systems were equipped with enough RAM memory to store all data it would maximally need, often resulting in over-allocation. [15] ThymesisFlow would allow a device to be provisioned for an average amount of memory, and only borrow extra remote memory when it requires it. IBM foresees this system to allow for equipping a server with enough RAM for its nominal requirements, if the server needs more memory during an intensive operation it could dynamically hot-plug remote memory into local memory. As memory accounts for around 50% of server cost, this would mean a major cost saver for servers.

Important to note is that ThymesisFlow works on a cacheline level. When a memory instruction is dispatched to a remotely mapped node, ThymesisFlow will fetch a single 128-byte cacheline from the remote. Compared to a page fault architecture, cache line level memory access is faster and more fine-grained [16]. In page fault systems memory accesses triggered to non-local memory will trigger a page fault, to be handled by the operating system. ThymesisFlow does not work with page faults, but rather bridges cache miss information onto the remote OpenCAPI connection.

In this thesis we will also use the property of transparent remote memory to allow for faster and more efficient memory transfers across compute clusters.

2.7.1. Comparing ThymesisFlow to SMP and NUMA systems

In Symmetric MultiProcessing (SMP) devices all servers in the system share a common memory space. More importantly, all this memory has an equal access time. On the other hand in a Non-uniform Memory Access (NUMA) architecture local memory can be accessed more quickly than remote memory. The ThymesisFlow system is closest related to a NUMA system. Remotely mapped memory in ThymesisFlow has some extra layers of address translations required, and for now has 2 FPGA cards in-between which cause remote memory to have a higher latency. Looking at cache coherency systems. SMP guarantees all of the address space to be cache coherent, commonly described as all CPUs are in the same coherency domain. ThymesisFlow on the other hand purposely does not guarantee cache coherency as to omit the expensive coherency communication between nodes.

2.7.2. Latency of ThymesisFlow compared to local memory

ThymesisFlow is promised to bring quite a low latency increase [1]. Also the latency of memory is highly dependent on where the data is cached. The most simple form of a pointer retrieval from memory takes 4 CPU cycles when retrieved from L1D cache, while it 37 cycles + 64 ns when retrieved form RAM. ThymesisFlow was measured to take 800ns for a single flit to do a full round trip to remote memory. This is largely due to the FPGA stacks at the lender and borrower end. As ThymesisFlow remote memory is also cached in L1,2,3 caches, this 800ns should be compared against the 37 cycles + 64ns of local memory. At 3800Mhz 37 cycles come down to 74ns. Although the latencies are influenced by a lot of varying factors, this simple comparison gives an idea of how the latencies of remote and local compare. Note that we discard all caching in this comparisons. The Power processor does Instruction re-ordering to optimize memory accesses, and also pre-fetches cache lines when the processor is able to predict memory access patterns. Once either local memory, or remote ThymesisFlow memory is cached they will have identical latency access times.

2.7.3. Mapping remote memory to local address space

On top of ThymesisFlow some libraries have been developed which use the HW/SW stack. The *libtf-shmem* library allows for mapping remote memory into local memory. The library consists of a Linux kernel module, and some code run by the application. This library allows for mapping remote memory into local userspace memory. The memory is mapped into memory using the mmap syscall discussed in section 2.6. By using the native mmap syscalls we can use any instruction and point it at the mmap'ed ThymesisFlow memory and interact with remote memory. This is very powerful as neither the kernel nor the application need to do any address translation. All address translation is done by the processor TLBs, OpenCAPI, and the ThymesisFlow hardware.

On the memory lender side we first initialize the memory region to be borrowed. The address of this region is called *Effective Address* internally in Power architecture, or more commonly Virtual address. The Effective Address is then sent to a *Borrower node*, who needs the memory. The borrower tells ThymesisFlow to map the EA provided of the given remote node into its local Virtual Memory address space. Internally libtfshmem will initialize the ThymesisFlow hardware and will call the Linux *mmap* syscall to map the remote device into the application's virtual memory space.

2.8. ThymesisFlow Cache coherency (non-)guarantees

The OpenCAPI specification stands for Coherent Accelerator Processor Interface. Care should be taken on what exactly is meant with the coherent part of the specification. OpenCAPI is meant to be an accelerator interface within a single machine. The coherence part is for accelerators or devices plugged into the machine, it does not maintain coherency across for example multiple machines. ThymesisFlow has explicitly chosen to not make all CPU caches coherent across a cluster. ThymesisFlow bridges cacheline traffic, but not coherency traffic to remote nodes. This makes ThymesisFlow different from an SMP system where all memory is shared across all cores in a cache coherent fashion. The ThymesisFlow documentation states: [1]

> This design bridges directly processor cacheline traffic, without any further intermediate

> caching support, but not coherence traffic. Therefore it enables scale-up of memory resources from the perspective of the compute node, *but cannot be used to further expand its SMP domain with remote CPUs.*

For the use case for which ThymesisFlow was designed this is not a problem. Memory is permanently borrowed by a compute node which requires more memory, the lender which owns the memory has no intent to read or write to the memory. When writing to memory, only the caches of the CPU writing to memory are updated. Any other CPU caches which can access the piece of memory are not updated.

This has behavior and even the exact use case has been studied before [10]. The authors of this paper have also localized some issues with cache coherency when writing into remote memory, but an extensive look into more access cases is missing. An extensive look is necessary what exact access patterns make the system cache incoherent.

For an accelerator device plugged into a OpenCAPI port, there are multiple states in which a device can communicate on the bus. They differ on how the cache architecture is layed out, and which device has memory. In the case of ThymesisFlow, a CPU has an FPGA plugged into the OpenCAPI interface. The FPGA and CPU communicate in the *OpenCAPI C1 and M1 mode*.

2.8.1. Lender OpenCAPI mode

On the memory lending node the FPGA is connected with OpenCAPI using the C1 mode. C1 mode means that the FPGA itself does not have any local cache, but can issue requests through the OpenCAPI link to snoop the processor cache. Thus when remote requests to read the local memory come in, they are first checked against the local CPU cache before being bypassed on to the memory device. When writing from the FPGA into the main memory, the caches of the CPU will *NOT* be updated. In the OpenCAPI spec this is noted as *no-intent-to-cache* operations.

This becomes a problem when remote nodes want tot write into local memory through the FPGA. As the local CPU caches are not updated and are thus invalid. We do want to be able to write into remote memory, and we will solve this in a software coherent fashion. In section 3.8 we will explain more on how this is done.

To reiterate, reads are cache coherent. When a read comes in through the OpenCAPI attached FPGA, they are first snooped from the processor cache. If there are any writes pending in the CPU cache, these are the most up to date values, and are thus returned to the compute borrower node.

2.8.2. Borrower OpenCAPI mode

The borrower side of the link operates in a different OpenCAPI mode. The compute node interfaces with its FPGA in M1 mode. In this mode the processor will assign a memory address space to the device. When the processor tries to access memory assigned to the device, the transactions will be forwarded to the device to be handled internally. In the case of ThymesisFlow, these transactions will be transmitted to the memory lender's FPGA stack through the fiber link.

Similarly to the C1 mode, the FPGA signals that it does not have any local caches, and has no local memory. Any memory transactions handed to the FPGA from the CPU are not cached by the FPGA, and thus OpenCAPI enables no cache coherency logic.

Important to note here is that memory writes from the CPU do not immediately end up in the FPGA. Writes may be buffered in the CPU cache to be written later on. To ensure data written to an address space belonging to a certain device, the processor will need to explicitly issue flush instructions. In the case of ThymesisFlow this is especially important as flushing will guarantee the actual writing to remote memory.

2.8.3. How ThymesisFlow interacts with CPU caches

The processor itself manages several layers of caching, each of which is for a more broader scoped number of cores and processors. In the case of the IBM Power 9 processor there are the following

three cache layers:[12, p. 155, 165]

- L1 cache. 32KiB per core
- L2 cache. Per 2 cores, 512KiB
- L3 cache. 10MB per 2 cores, accessible by all other core pairs.

These caches act as a single cache from the view of ThymesisFlow, and their individual properties are quite hidden to the implementation. This thesis does use these numbers to make correct benchmarks which are substantially bigger than the cache size to ensure data is actually written through to the underlying memory. More on that in section 5.1.

For ThymesisFlow this means that a single piece of memory on the lender side, can be cached on both the borrower and the lender without cache coherence in between. The CPUs on the lender and borrower have no visibility of each others caches and can therefore not even know another CPU has cached memory of its memory. ThymesisFlow suggests applications to solve these limitations by applying software coherency systems. [1]

There are three types of CPU caches active in a fully disaggregated system. Every node is able to read and write into every other nodes' shared memory.

- CPU cache of the node which ”owns” the memory
- CPU cache of the node writing to the lender's memory. This may also be the lender itself.
- CPU cache of any other node, remote to the ”owning” node

To clearly lay out how these coherency limitations impact a fully disaggregated system, we will analyze on a per case basis how it goes wrong. In the Methodology section we will discuss ways how to solve these coherency issues. We can describe several ways this thesis interacts with memory:

- A lender writes into its own memory. This is the easiest case as OpenCAPI guarantees coherency. Writes may be held in the CPU cache, before being written to memory. When a borrower reads form this memory, OpenCAPI will first snoop the lender's CPU cache, and then the memory if the cache does not contain the correct address. But, when a borrower has cachelines of the memory the lender has written to, these are not updated, and are thus invalid. When a borrower reads the lender's memory, the reads will have a hit on the borrower's local CPU cache, and the reads will never hit the ThymesisFlow bus. The borrower will thus read incorrect old values. An example of this is discussed in subsection 3.7.1.
- A borrower writes into a lender's memory, a lender reads that memory. In this case the borrower writes its data into the ThymesisFlow mapped memory. The issue here is twofold. First of, the writes from the borrower are not guaranteed to be written to the lender's memory. The writes may be held in the borrower's CPU cache. Second, the writes will be done on the lender's memory, but they won't be written to the lender's CPU cache. When the lender has cachelines of the memory the borrower has written to, these will be outdated. An example of this is discussed in subsection 3.9.2
- A borrower writes into lender's memory, another borrower reads that memory. The same holds for the previous point. When the writing borrower writes to the lender's memory, the CPU cache of another borrower is not updated. The same two problems occur as the previous point.

3 Methodology

A system has been designed which extends Arrow to not only manage locally created objects, but also allow for disaggregated Arrow objects. The extensions allow for cache coherent object creation, which are readable by other nodes through ThymesisFlow. The impact to the user Arrow API is minimal, most of the logic to make the system cache coherent is abstracted away. Calls were added to allow an application to send objects to other nodes in a cluster, and even allow for objects to span the memory of multiple nodes.

Each node has a so called *Orchestrator*, which manages the integration with ThymesisFlow. The orchestrator is responsible for transferring table descriptors to other nodes, and has waiting logic to allow to wait on the whole cluster to come to the same execution point. Initialization of the ThymesisFlow memory maps is done inside the orchestrator. Furthermore the orchestrator manages configurations to allow for seamless access to the ThymesisFlow hardware. Finally the orchestrator is also responsible for a special feature which allows for tables of multiple nodes to be combined into one. This is called a *HugeTable* in this thesis.

As we will discuss throughout this thesis, ThymesisFlow alone does not suffice to communicate in a cache coherent manner. To ensure coherent communication a separate communication medium needs to be employed. gRPC was chosen for this. Every node has a gRPC server for incoming messages, and every node is a client to every other nodes' server. The communication protocol is defined using Protobuf definitions.

Furthermore many modification were made to change the memory manegement in Apache Arrow. In a local situation, Arrow uses the MemoryPool abstraction to allow for allocations in several allocation schemes. In the remote sense this was extended to allow MemoryPools to allocate memory in remote memory. Allocation calls by MemoryPools are passed to an Allocator. These allocators were also modified to contain the software cache coherency protocols, and call the low-level malloc, free and realloc methods.

The components described in this chapter fit in the system diagram in Figure 3.1.

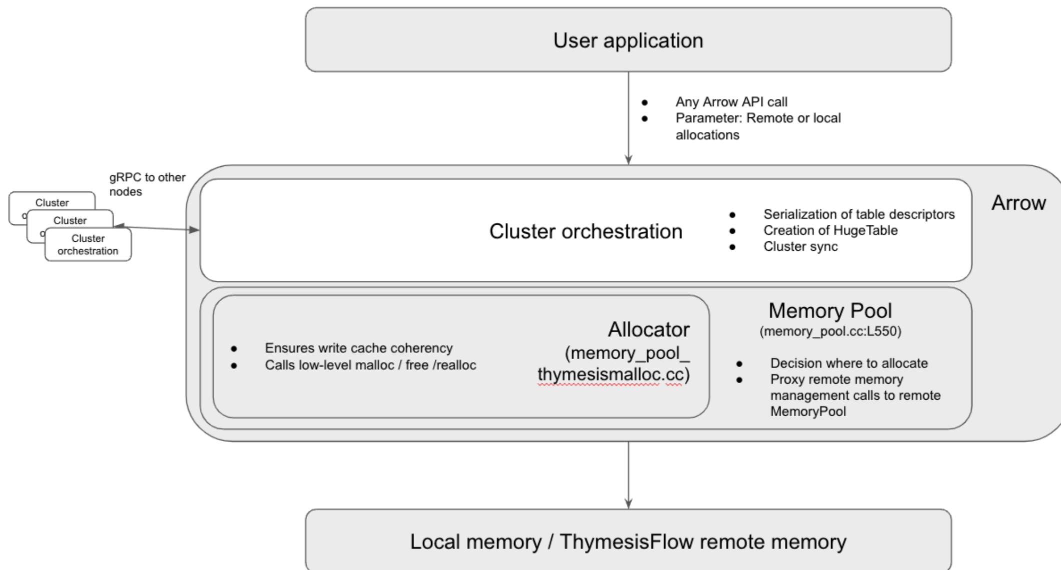


Figure 3.1: Designed system abstraction layers

3.1. Inter-node communication using gRPC

Quite some information needs to be exchanged between nodes besides the raw memory provided by ThymesisFlow. We will use ThymesisFlow to transfer the Arrow data, while we use gRPC [17] for orchestration, transferring state information, and structure of the Arrow data. Important to note here is that we do not use gRPC to transfer the actual data stored within a table, merely the *table descriptor* of Arrow objects as described in subsection 2.1.5.

gRPC works with a server client model. A server can accept connections from multiple clients. For this thesis every node will need to be able to communicate with every other node. This is achieved by starting a gRPC server on every node, and having every node be a client to every other node. When a node A wants to communicate with a node B, node A will use a local gRPC client to connect to the gRPC server of node B.

The functions the gRPC are:

- SendRecordBatchMD: Sends a table's structure and references. Does not send the actual contents of the table
- SendState: Sends the current application state. Used for synchronizing all nodes in the cluster
- Malloc: Request a block of memory of the server, returns a pointer to this address. This address is accessible thorugh the ThymesisFlow link.
- Reallocate: Extends or shortens the previously malloc'ed block of memory. The server handles a possible memcpy.
- Free: Releases a previously malloc'ed block of memory.
- Ping: Simple round trip to verify a server is online.
- SendTableFull: Sends an Arrow table with the internal data arrays. Used for comparing against a zero-copy transfer with SendRecordBatchMD

As will be discussed later on, the gRPC communications are only required during the initialization and destruction of objects. After initialization no expensive gRPC calls are needed.

The message format is defined in protobuf definitions. Protobuf is a "language-neutral, platform-neutral extensible mechanism for serializing structured data" [18]. Protobuf is used to define the communication protocol for the data sent over the gRPC link. An example protobuf definition for serializing an Arrow RecordBatch is given below:

```
message RecordBatchMD_v2 {
  message Field {
    message Buffer {
      // uint32 capacity = 1; // As we will not be writing from remote, this is not necessary for the other party to know
      int64 size = 2;
      uint64 pointer = 3;
      uint32 level = 4;
    }

    bytes name = 1;
    uint64 type = 2; // static_cast<Type>(data[0])
    int64 length = 3;
    int64 null_count = 4;
    repeated Buffer buffer = 5;
  }

  uint64 rows = 1;
  uint64 start_index = 2;
  repeated Field field = 3;
}
message SendRecordBatchMDReply_v2 {
  bool success = 1;
}
```

To broadcast a certain message, a node will have to connect to every other node and call a gRPC method on that node.

3.1.1. Synchronous RPC

Currently the RPC methods are written using the synchronous API. Synchronous being that the caller RPC method will only return when the callee has fully executed the RPC method. This is not very efficient as this means that when we want to broadcast a piece of information we will have to serially visit every client, and wait for the RPC call to return for every one of them.

For example, when we want to send a RecordBatch table descriptor packet (without data), we would have to visit every node and call the RPC send command serially. Only one node will be able to process the packet at the same time. gRPC does have asynchronous RPC calls, but implementing them is out of scope for this thesis. Also as this overhead is only present during the initialization of the Arrow objects, no gRPC communication is done during read writes of the actual data, the overhead incurred by serially broadcasting packets is thus $\mathcal{O}(1)$.

3.2. How to transfer Arrow data

We require the ability to read and or write data across a compute cluster through the ThymesisFlow link. This will be done without copying the data itself, only communicate the location and metadata of an object to another node. We call the Arrow object without the actual data, only the references to the data and the structure of the data, the *table descriptor* of an Arrow object. There will be many references to the wording *table descriptors* in this thesis.

Assuming the data buffers are accessible by all nodes, nodes need only the *table descriptor* to be able to read the underlying data buffers. The table descriptor for example contains a pointer to a memory buffer, this pointer will lie within the ThymesisFlow remotely mapped memory. A user application will deserialize a table descriptor into a proper Arrow object, with this pointer to remote memory. When this pointer is accessed, ThymesisFlow will proxy the memory traffic to the remote node. By default Apache Arrow does not have serializers available to extract the *table descriptor*, without the data, into a transmittable buffer. This will be designed and created.

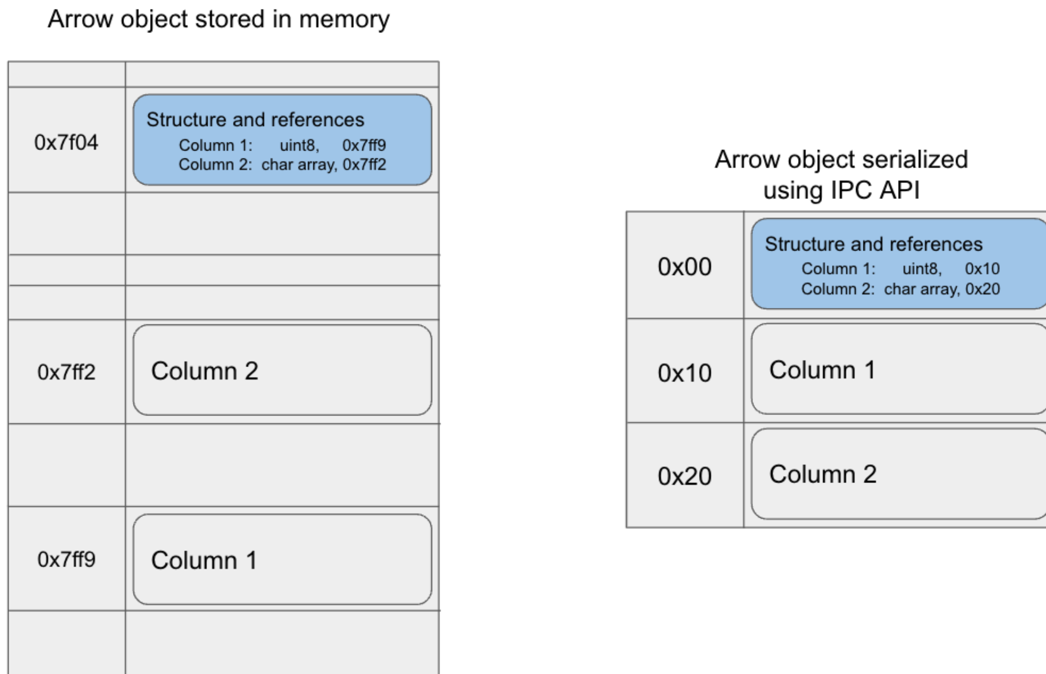


Figure 3.2: Arrow IPC serializer. From a table stored throughout a process memory into a location independent packet

3.2.1. The Arrow Inter-Process Communication API (IPC)

Apache Arrow does contain methods for serializing a full table, including the data itself into a buffer, this is done using the *RecordBatchWriter and RecordBatchReader* IPC methods. These are for example used if we want to copy an object from one node to the other through an ethernet connection, when we want to save objects to disk, or when we want to share an object through shared memory mappings.

In an IPC generated buffer the previously described *table descriptors* are stored, and also the data buffers themselves. Important to note here is that the Arrow IPC packet table descriptor use offsets relative to the start of the data packet, while a regularly stored arrow table may be stored throughout an applications memory, with addresses local the the process which created it. In Figure 3.2 we seen an example Arrow RecordBatch. On the left the object stored throughout memory, the table descriptor stored in compiler allocated memory, and the column buffers placed by the operating system malloc calls in heap memory space. While on the right we have an IPC packet, which is memory position independent.

These IPC API calls are not zero-copy. The IPC writer calls will copy all data into a new buffer. This is done because the new buffer may then reside inside a shareable location, and will contain offsets which are valid for every process reading them with a potentially different memory layout. Let's take for example a shared memory mapping to send an Arrow RecordBatch from one process to another process. See Figure 3.3. Both process have a shared memory region which is visible to both of them, but this region is potentially placed in a different memory address by every process. Because the IPC packet placed in the shared memory region is position independent, both processes are able to read the IPC packet without requiring any offset calculations. Also both processes are able to read the packet without having to copying it again to local memory. Thus when sharing a RecordBatch using IPC, Arrow does a single copy operation into the shared memory region, after which other processes are able to read that data without first having to copy it locally.

The IPC logic in Arrow is a possible candidate to be used. It could be modified to not serialize the data, only the table descriptor.

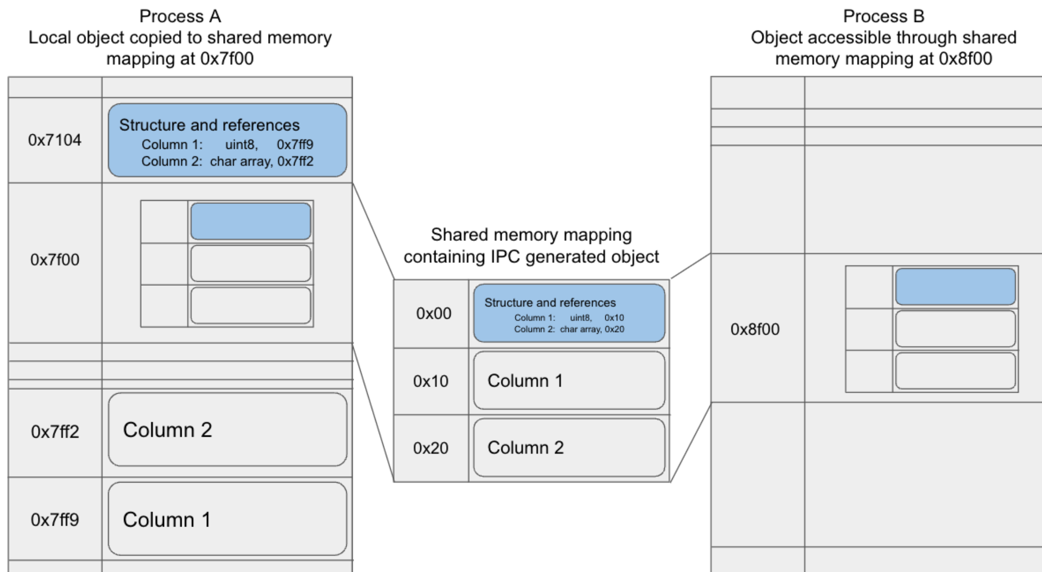


Figure 3.3: Arrow IPC object accessible by two processes through shared memory mapping

3.2.2. Considerations in modifying Arrow to serialize table descriptors only

To serialize only the *table descriptor* of an Arrow object we could do one of three things:

1. Modify the IPC API of Arrow to not include the data, only the table descriptor
2. Store the *table descriptor* next to the actual data buffers.
3. Write a custom visitor class which traverses the Apache Arrow structure and generates the *table descriptor* packet.

As a guideline the solution which has the least amount of impact on the existing Arrow code base should be chosen. We also want to keep the existing Apache Arrow Format specification fully intact, without any changes. Finally, the application facing API of Arrow used by user applications should not be modified, merely extended, to allow for existing Arrow applications to use the ThymesisFlow platform without any rewrites.

Modifying Arrow IPC

The obvious choice to extract table descriptor data is to modify the IPC API. The IPC API contains code for serializing a full table including the data buffers. A modification to this would mean we keep the table descriptor serializer functionality, but not have them change the pointers to the data buffers to relative offsets, but keep them as absolute offsets.

However, the code for the IPC API is vast and complex. Arrow supports some very complex data structures such as nested lists, dictionaries, and mixed type arrays. The functional component to go from a shared_ptr of a RecordBatch to a standalone buffer which can later be used to initialize a new RecordBatch class instance, is not modular and hard to extract. For the purposes of this thesis we want to create a prototype, and do not require to support all Arrow features.

Table descriptor allocated near data buffers

Another possible method considered is to store the table descriptor of the object in the shared memory space immediately during the creation of an Arrow object. Looking at Figure 3.2, Arrow could be modified such that the data is not spread out throughout the memory address space of the application, but rather immediately placed inside the shared region. When an application wants to read a table at a certain location, it only needs to be pointed at the location where the table descriptor is stored. The references then point to the addresses where the data buffers are located inside the shared memory region.

Although this is possible for the data buffers, this becomes more difficult for the table descriptors. As discussed in subsection 2.1.5, Arrow by default stores everything created in compiler allocated storage. For example the high-level RecordBatch class is instantiated by the user application. This may be done statically on the stack, or it may be done using the *new* operator. The stack is not a portable way of exchanging data with other processes. As the location where data is stored is fully dependent on the call-stack of the user application. By using the "new" operator, memory is allocated on the heap, the location of which is static from the moment it is allocated. This brings more opportunity, but does limit the user applications to not use statically allocated Arrow objects.

An issue with sharing the table descriptors with each process is that Arrow makes extensive use of shared_ptr objects. A SimpleRecordBatch for example contains a vector of shared_ptr to ArrayData instances, which in turn contain a vector of shared_ptr to Buffer objects. These shared_ptr helpers are not designed to be used for across multiple applications. They keep an internal counter which keeps track of how many references to that pointer exist in the run time of a single process, they are not able to keep track how many references to that object are across multiple processes.

Another issue is that Arrow is not designed to create IPC formatted structure and reference data dynamically when creating objects. Arrow only has code for builders to create a RecordBatch, or has methods to instantiate a RecordBatch statically in class instances. The only way to create the IPC payloads is to first create the RecordBatch, and then convert that into an IPC payload. Rewriting arrow to immediately create the IPC payloads using builders is a considerable redesign, which would take up a lot of time. Arrow *does* have code for reading without copying though. An IPC created object contains the table descriptor information next to the actual data buffers, a reading process is able to create and instantiate valid Arrow objects from this IPC packet without copying any data or table descriptor information.

Creating a custom visitor class to serialize table descriptor information

The final solution used for this thesis is a combination of the first and second approach. The idea is used of the second approach to allocate the Arrow data buffers immediately in a shared buffer space. This buffer space is readable and writable by all machines in the compute cluster through ThymesisFlow. Secondly the idea of the first approach is taken, in which all the builder and instantiation code of the table descriptor data of an Arrow object is left intact. Only when the application needs to "zero-copy" transfer an object the table descriptor is serialized and sent to another process. The result is that the data itself does not need to be copied as it is immediately placed in a mutually accessible shared memory space. Furthermore the table descriptor can be created using the existing Arrow code, when another node wants to access the table it only needs to serialize, send, and deserialize the structure and reference information. All the pointers to the data are kept, as they are also accessible by another node through ThymesisFlow.

This ensures no thorough redesign of Arrow is necessary, as the general structure of the code base to instantiate objects is kept intact. Furthermore the compiler is still able to optimize method calls as it was able to do before.

For the serializing of the Arrow table descriptors into a standalone C data structure, code of the Fletcher project was used. Fletcher employs an Arrow visitor class which is able to serialize Arrow RecordBatch table descriptors, to be used for a connected FPGA accelerator. [19] This code is extended to allow for some extra data types, include code to convert the serialized structure into a protobuf object, and finally add code to deserialize the protobuf packet into a RecordBatch instance.

3.3. Accessing remote memory using ThymesisFlow and mmap

In the previous section a description was given on how Arrow could be modified to serialize and deserialize table descriptor information. In these table descriptors there exist pointers to underlying memory buffers. These memory pointers are not changed, and should be valid on any node which accesses them.

ThymesisFlow is used to make these memory pointers, and thus the underlying memory, accessible by any connected node. Specifically remote memory regions are mapped into the user application's

memory region. In Linux this is done using the mmap syscall. ThymesisFlow provides a memory device, which any application can then map into its memory using the mmap syscall.

The mmap call allows for the *MAP_FIXED* syscall flag. This flag tells the kernel the pointer passed to it should be interpreted as an exact location where the memory should be mapped into memory. This allows us to map a remote ThymesisFlow memory region to the exact same location in every node. A memory address pointer in one node, points to the same data as that pointer in another node. For example when a local Arrow object is instantiated at address *0xff*, accessing *0xFF* on another node will access this same Arrow object. ThymesisFlow will proxy any requests to this region to the correct remote memory.

ThymesisFlow and its libraries were modified to use the mmap syscall instead of the posix_memalign function to allow for passing the MAP_FIXED flag.

The procedure for memory allocation will be as follows:

1. ThymesisFlow is initialized, a pre-exchanged EA is passed to ThymesisFlow, which makes the remote or local block available for other nodes to map. The EA is an address valid in the lender's user space address space, passed to the borrower through grPC.
2. Using mmap with MAP_FIXED flag, the ThymesisFlow memory is mapped to userspace memory. With the MAP_FIXED flag the kernel maps this block into user memory at the exact specified address.
3. A remote borrower ThymesisFlow instance is given this EA, and initializes the connection with the lender side.
4. Again we use mmap with MAP_FIXED flag to map the remote memory at the exact save EA as the lender.
5. Both the lender and the borrower now have the allocated block of memory mapped ath the same location.

3.4. How and where to allocate Arrow objects

Apache arrow has several malloc implementations with which it allocates memory. By default it uses the jemalloc library, but can also use the built-in libc malloc allocators. Internally Arrow has some abstractions to handle the low-level memory allocations. On the highest level we have *Memory Pools*. Memory Pools are C++ classes which are usually globally instantiated, and are a wrapper around the *Allocator* classes. Memory Pools are meant to keep track of statistics and take care of aligning memory addresses. In subsection 3.8.2 a description is given what needs to be extended to also allow for allocating on remote nodes.

Next we have *Allocator* classes. These are wrappers around the library malloc end other memory management calls. The Allocator simply calls the library, and will convey any errors into Arrow Error objects if needed. Allocators can also be used to implement re-allocation calls if the underlying library does not support it.

For the integration with ThymesisFlow we modify the behavior of the MemoryPools and Allocators. No longer will they allocate at random addresses defined by the library, kernel or libc. The allocators will have to allocate within the shared address space mapped by ThymesisFlow, so that remote nodes can access it through their own ThymesisFlow mapped regions. As a node has a memory region for every remotely mapped node, allocations need to take place in one of these regions. This is done by creating a new ThymesisFlow supported Arrow *Remote*MemoryPool. The new RemoteMemoryPool allows for user applications to tell Arrow in which nodes' memory it wants to store objects and where Arrow should allocate objects. A node has a RemoteMemoryPool for every remote memory it has access to. When a node wants to allocate Arrow objects in the memory of a node, it can simply pass in the specific RemoteMemoryPool of that remote region.

Because malloc and its family of functions by default do not allow for custom regions to allocate within, the only thing an application has control over is the brk and sbrk sycalls, a custom memory allocation

Figure 3.4: 64-byte alignment of Arrow objects cause cache problems when ThymesisFlow is used.

system is to be developed. [1]. This custom malloc implementation allows for applications to define new memory regions, in which malloc/free/realloc can be called. Specifically the implementation should allow Arrow to have many zones, and malloc should have a parameter to define in which zone allocations should be done.

3.4.1. Cache line alignment of Arrow objects: 64-byte to 128-byte

Arrow by default aligns all data buffers to an 8 or 64-byte address. This is done to allow for efficient SIMD operations on Intel processors. To allow ThymesisFlow and its caching operations to be coherent we modify this requirement from 64-byte alignment, to 128-byte alignment. The PowerPC architecture has 128-byte cache lines, and the ThymesisFlow system uses 128-byte cacheline aligned data packets to communicate over the OpenCAPI busses. Due to the flushing systems described in section 3.5, it is not allowed to have different Arrow objects in a single 128-byte cacheline. If this were to happen, cachelines of untargeted objects may become invalid, and thus cache incoherent.

Let us take the situation in Figure 3.4, with 64-byte alignment of data, on a PowerPC with 128 byte cache line sizes. On the left situation, before write, identical data stored is stored in both the cache and memory. On the second half of the cache line though, different data is stored in the cache line compared to the underlying memory. This can happen if the memory in question is written to by another machine through the ThymesisFlow link. The systems in this thesis prevent coherency inconsistencies for shared data, it does not guarantee coherency for all non-shared data. The second half of the cacheline may for example be a local object of a remote machine, which the local machine has no reason to access. In the ”before write” this is not a problem, as the local machine will not read or write the data. However, when the local machine writes, denoted as *After write*, to the first half of the cache line, we do have a problem. If the caches are now flushed, the ”33 33 33 33” values will be correctly flushed to memory, but also the incorrect values ”EE EE EE EE” will be written.

This issue can be worked around by extending the alignment reuirement of Arrow objects from 64 bytes, to 128 bytes. This ensures different objects are never stored in a single cache line, instead every object has its own cache lines.

Increasing the alignment requirement from 64 to 128 bytes does not impact the normal operation of Apache Arrow, as every 128 bytes address is also 64-byte aligned. Furthermore the Arrow format specification states objects should be aligned with *multiples* of 64-bytes. Care should be taken though with objects created by non-modified Arrow libraries. A non-modified library as per this thesis may allocate on 64-bytes, which may fall inside a 128-byte cacheline.

3.4.2. Resulting Arrow allocator flow

The resulting workflow for allocating a coherent Arrow object available for sharing is thus as follows:

1. Map shared memory on all devices to userspace memory using ThymesisFlow and mmap, as

[1] Mac OS X does support changing the allocation behavior using so called *zones*, using the `malloc_zone_malloc` function we can then allocate inside these zones. Mac OS X does NOT support placing these zones at a specific location though.

described in section 3.3

2. Instantiate the allocator data structures in the mapped memory region. We are now able to call malloc/free/realloc for this memory region
3. Create the Arrow MemoryPool with corresponding Arrow Allocator objects. These point to the memory region, and are saved for retrieval by user application.
4. If a user application now wants to allocate in a certain region, it can retrieve the MemoryPool of the corresponding memory region. This MemoryPool can then be passed to any allocating Arrow API, which will execute allocations in the correct memory regions.

3.5. From partial hardware coherency, to full coherency employing software coherency

In section 2.8 we discussed what is cached, and what is not cached in the ThymesisFlow system. To reiterate, on both the lender and borrower we have active CPU caches for both local and remote memory. This means that the lender, who owns the memory region, potentially has cached data in its CPU cache, but also that any borrower attached to the lender's memory my have cached data of the lender's memory in its CPU cache. The challenge is to have not only the local CPU cache coherent, but also any remote cache coherent.

A solution not explored in this thesis to disable all CPU caches of the remote memory region. The whole benefit of OpenCAPI/ThymesisFlow is that it is able to use the CPU caches for both local and remote memory, disabling the CPU caches would give an unreasonable performance hit.

3.5.1. Using Immutability of Arrow objects for coherency

For this thesis we need to make the caches of both the lender and borrower cache coherent. When a borrower writes Apache Arrow data into the lender's memory, the lender should be able to use it in a cache coherent fashion, and vice versa. Apache Arrow's data format has a property which is exceptionally well suited for this requirement. *Apache Arrow objects are immutable once created*.

This immutability comes in quite handy when making the system coherent. Coherency becomes an especially difficult task when the data to be kept coherent is continuously changing. When the objects are immutable, the system needs to be made coherent only once during initialization. After the initialization, caches can not become out of sync with the internal memory as the data memory is never changed. Another benefit is that we only need to do expensive coherency operations once, at initialization. As flushing and invalidating of caches can be a very expensive operation, we do not want to flush too often.

3.5.2. Making remote and local caches coherent with flushing, without invalidation

In subsection 2.8.3 several cache incoherent cases were discussed. In this section we will handle the first case where memory is written by only the node which owns it. Later on in section 3.9, the more difficult situation is described in which data is written by the borrower, into the lender's memory.

A node which writes into its own memory will, by default in Linux with write-back pages, have some of the written data be held in the CPU cache. This system ensures that write instructions can be fast, as when there is more CPU time, the writes held in the CPU cache are written through to the underlying memory device. For this case it is actually less important where the data is held, in CPU cache, or written through to memory. As any incoming remote memory requests, will snoop through the OpenCAPI link the CPU cache before doing a memory lookup.

Care should be taken with caches in remote nodes though. If a borrower node has cachelines of a lender's memory, the borrower CPU will never be able to fetch the remote memory, as the local CPU cachelines will always be returned. Also if a remote borrower node has any pending write cache lines,

these may overwrite a lender's memory once written. Thus when a lender wants to write to its own local memory, it needs to first ensure all other borrower CPU caches are emptied.

Emptying CPU caches however, is not a trivial task. CPU caches are usually handled by CPU hardware, and are not meant to be influenced by applications. The Power Instruction Set Architecture (ISA) has two instructions which are relevant here. First of there is the `dcbi` instruction, *Data Cache Block Invalidate*. This instruction tells the processor to empty the cache line which the given address is contained in. This instruction however has been obsoleted in Power9 [12] due to a cache coherency issues in the processor design. The only thing an application can do is cache *flushing*. Flushing meaning here that a pending cache write is written to memory, and any other caches discarded. The *dcbf* instruction is used in this case. This instruction does the following: [20]

> copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid

More about the flush instruction chosen and the possible alternatives can be found in the Implementation chapter, in section 4.8.

The dcbf instruction however does incur a performance hit compared to invalidating. When a lender wants to write into its own memory it would have to first ensure the caches of *all* borrowers is flushed. As it is not possible to first write, and then tell all borrowers to invalidate their cache lines.

The procedure therefore for a lender to write into its own memory, to be shared, can be done using software coherency, or weak consistency protocols [2]. Concretely it should look like the following.

1. Allocate a memory block, using the allocators described before
2. Before starting the write into a memory region, the lender tells all borrower nodes to flush any pending writes. This ensures no writes are hanging, and any reads afterwards are re-fetched from memory, now containing the newly written data.
3. Lender writes to the memory region
4. Any reads from any CPU are now cache coherent. Because all CPU's have flushed their cache-lines, any data access will re-fetch the memory and repopulate their CPU caches.

3.5.3. Prevent instruction re-ordering to move flush before write

Threads in the Power 9 processors are not executed exactly simultaneous. When a thread stalls, other threads will get room to execute. This ensures that instructions slots in the processor are optimally used. The Power 9 processor uses Simultaneous MultiThreading (SMT) for this. When a thread encounters instructions which will take a long time, the CPU may even start re-ordering the instructions of the process to ensure that when a instruction occurs which it cannot immediately execute, the pipeline of the thread does not stall. The processor analyses which instructions are dependent on which instructions, and is able to move instructions with no dependencies on instructions still to be executed, to be moved earlier in the pipeline.

For example, take the example below. Loading data from memory is slow and may cause a stall in the CPU pipeline, this happens in instruction 1 and 2 below. Before instruction 3 is able to be executed, it is dependent on the results of instruction 2. But the value of A is already available, and the instruction 4 is already able to be executed. The Out-Of-Order unit of the processor may then move instruction 4 to place 3, allowing for the A increment instruction to be executed before the B increment instruction.

1. Register A is loaded from memory
2. Register B is loaded from memory
3. Register B is incremented
4. Register A is incremented

This re-ordering of instructions is done on a hardware level by the processor, and the program is usually not affected by this. The processor ensures that instructions it executes earlier than they occur in the program, are not dependent on the instructions not yet executed. In the case of memory sharing with

other nodes we do need to be careful here. In the previous section we discussed how after a write to memory, the data is flushed from cache to memory. Let's take a simple assembly program to illustrate this:

```
addi    rD, 0, AA        ; 1. Set register rD to value AA
stb     rD, 10(0)        ; 2. Store rD into ram at address 10
stb     rD, 11(0)        ; 3. Store rD into ram at address 11
addi    rA, 0, BB        ; 4. Set register rA to value BB
dcbf    10               ; 5. Flush cache block containing address 10
```

Here the processor will recognize that instruction 2 depends on instruction 1, and will thus keep the order of instructions intact. The fourth instruction however is not dependent on the value of instruction one, two or three. This means that when the processor encounters this instruction sequence it may move the fourth instruction before the third instruction. This will ensure that the processor pipeline does not contain a stall, and will reduce the amount of time the processor is doing nothing. While the second instruction is waiting for its result, it can execute the fourth instruction.

The hard part happens when the processor encounters the dcbf instruction. The dcbf instruction flushes a cache line at a certain address. This instruction has an input of address 10, but executes on the full cache block which contains address 10. A cacheline is 128bytes long, and a dcbf instruction will flush the whole 128 bytes from cache. The processor only sees the address 10, and will assume that the dcbf instruction is not dependent on the output of instruction 3 with address 11, only on the output of instruction 3 as it operates on address 10. The Out of Order execution engine of the processor may then move the dcbf instruction before the third instruction, causing the cache line to be flushed before the whole cacheline has been written to. Resulting in a not fully flushed cache. This is a problem, as we want to flush all data written, not have the processor move our flush instruction before everything has been written to.

This is solved by not only calling the dcbf instructions when we want to flush, but also calling a sync instruction before and after the flush. The sync instruction is called a *Memory Barrier*, it disallows the processor to move instruction across the barrier. A sync instruction is called before a flush to ensure the flush is not moved before a memory write to that region, and a sync instruction is called after the flush is done to ensure any future reads are not moved in between flush instructions. The resulting assembly becomes:

```
addi    rD, 0, AA        ; Set register rD to value AA
stb     rD, 10(0)        ; Store rD into ram at address 10
stb     rD, 11(0)        ; Store rD into ram at address 11
addi    rA, 0, BB        ; Set register rA to value BB
hwsync                   ; Memory barrier to force flush *after* write
dcbf    10               ; Flush cache block containing address 10
```

3.6. Huge Tables: Spanning single table across all nodes.

As described in subsection 2.1.4 Arrow allows for creating columnar data, called Table, which has columns which are non-contiguous. A table contains ChunkedArrays as columns, which again themselves contain multiple Arrays. We can use this property to do quite an interesting abstraction. We can make tables which span multiple nodes. The HugeTable will contain columns, of which parts of the columns be stored in different machines. We call such a table a *Huge Table*. A huge table may be useful for:

- Tables may be bigger than the memory of a single node
- Faster initialization. Every node writes their own local chunk of the table, combined into a single huge table
- Distributed access patterns done natively. Any of the nodes are able to index the huge table, Apache Arrow will find out which array the index is stored in, ThymesisFlow will dispatch the request to the correct node's memory.

3.6.1. ChunkedArrays spanning multiple nodes

As stated before ChunkedArrays are non-contiguous arrays. Meaning that parts of the arrays may be stored at multiple locations in memory. We can use this property to store parts of an array on one node, and other parts on remote nodes. This will result in a single ChunkedArray, which is distributed across multiple nodes with a single shared memory space layout. A huge table has columns consisting of such ChunkedArrays. Any built-in Arrow compute functions which support ChunkedArrays, will natively support these disaggregated ChunkedArrays.

3.6.2. Memory translation done natively

The interesting thing here is that an application does not need to have any knowledge where data is stored. It will simply access memory referenced by the ChunkedArray somewhere in userspace address space. If the memory address is a remotely mapped address, ThymesisFlow will fetch the data from remote memory, or maybe even a locally cached copy of the remote data. All address translations are done by ThymesisFlow, and the local CPU caches are still active and can be populated with remote memory. The power here is that both Arrow and the user application have no knowledge of what memory is remote, and what is local. Load and store instructions are transparently either served from local memory, or retrieved form remote memory in a low latency fashion. This is a big plus, as the processor caches are still used for caching even remote memory, any cache prefetching logic of the processor also works on remote memory, and the out-of-order execution of the processor understands instruction dependencies on remote addresses.

3.6.3. Instantiation of huge tables

To initialize a *huge table*, we first create RecordBatches on local nodes. These are then broadcast by sending the RecordBatch table descriptor as described in subsection 2.1.5 to all other nodes. After all RecordBatch descriptors have been sent to every node, every node will combine these RecordBatches into a local instance of the huge table. Every node will have a local table structure which references the memory on either local or remote memory.

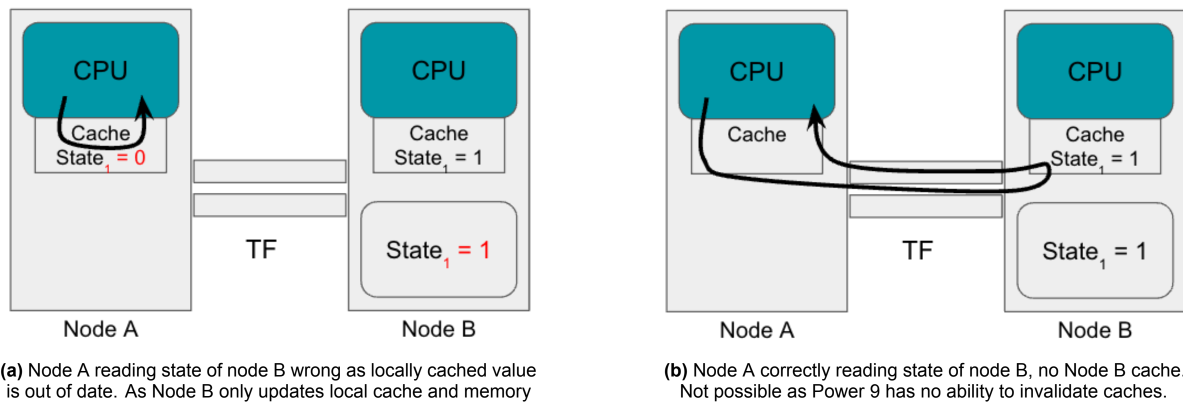
Interesting here is that the table structure contains the exact same information on every node. Every Array references the same memory addresses, a pointer one one node pointing to local memory, is also valid on a remote node. No pointer dereferencing occurs, no offsets calculations or conversions, or anything else happens here. As described in section 3.3, every piece of memory is mapped to the exact same location on every node to prevent offset calculations when accessing memory.

3.7. Synchronizing application state across a cluster

To ensure that the nodes in a cluster are synchronized, a state synchronization system is implemented. The system is able to have nodes wait on each other to reach a certain state in the program execution flow. The system should be able to guarantee that every node is in an expected stage, and that not computations are done which disrupt the software cache coherency protocols described in section 3.5. More specifically we want to ensure that:

- Arrow objects are not accessed before the objects have been made immutable, and the proper cache flushing operations have been executed
- Allow for collect stages in the program, allowing for creating tables spanning multiple nodes as described in section 3.6.
- Application code can wait for other nodes to reach a certain state to allow waiting for data to become available.

The synchronization system should have a low as possible overhead and should be resistant to race conditions. An obvious candidate for a high speed connection would be the ThymesisFlow link, as it has a low latency. However, as ThymesisFlow does not itself guarantee cache coherency, and the Power 9 processor has no way of invalidating cache lines only flushing cache lines, there is a problem with this.



(a) Node A reading state of node B wrong as locally cached value is out of date. As Node B only updates local cache and memory

(b) Node A correctly reading state of node B, no Node B cache. Not possible as Power 9 has no ability to invalidate caches.

Figure 3.5: State synchronization with ThymesisFlow

3.7.1. Why ThymesisFlow shared memory is not used for state synchronization

Let us describe an example system which uses the ThymesisFlow shared memory to communicate states to all other nodes. Every node allocates a local memory structure in a remotely readable ThymesisFlow memory region. This structure would contain the state the application on that node is in. Every time the application changes state, it writes its state into the memory structure, it polls every other memory region through the memory region, and waits for all structs to match its local state.

The problem here is the lack of cache coherency guarantee in reading remote memory in ThymesisFlow. When a node reads memory from a remote node there are several caches in between. Namely first the local CPU caches, then the remote CPU cache, and then when no cachelines match, the memory is read from remote memory. In the case of the example system, the node which writes into local memory to update the structure, would update its own CPU cache, as illustrated in Figure 3.5a. But it would not update the other caches across the system which may have the struct cached in their own CPU cache. When a remote node tries to read the status structure, the processor will first check its own CPU cache, which still contains the old state structure.

A possible solution to this issue is to use cache invalidation logic on the remote nodes. Every time a node wants to read remote memory, it would first invalidate its own local cache lines, then read the state structures from the remote node. This invalidation ensures that the data is fetched from the remote node through the ThymesisFlow link. This would look something like Figure 3.5b. As discussed before however the invalidate instruction is not available. The only instruction we can use to remove cache lines is the data cache block flush (*dcbf*) instruction. This instruction flushes the specified memory block from cache if it is cached. In the case of the example system we cannot use the dcbf instruction, as it would mean the old state of the cache is flushed through the ThymesisFlow link, and overwrites the newly written state.

Therefore we choose to use the gRPC link as a synchronization method, as to prevent cache coherency issues.

3.8. Allocating Arrow objects on remote nodes

Up until now Apache Arrow objects are created by a node which writes solely to its own local memory, and then shares the location of the data with other nodes. We would however also want nodes to create Arrow objects in remote memory. This would allow for example a single node in a cluster to read a big table from hard disk storage, and write this table in chunks to every node in the cluster. Or it would allow for simply having one node contain the data, and not having to send it over a slow ethernet connection to other nodes.

This gives rise to a cache coherency issue with the way caches are updated in the ThymesisFlow and OpenCAPI specification.

3.8.1. Malloc on remote memory

To allow for data to be written to remote memory, a remote memory space needs to be reserved. This is done using the malloc family of libraries. The malloc implementation described in section 3.4 is used for local allocations, and uses a linked list besides the allocated regions to store data about allocated regions, sizes en pointers. Because this implementation stores its data in the same memory regions as the data, the same coherency problem occur as the state synchronization systems described in subsection 3.7.1. A remote application has no way of reliably reading remote data through the ThymesisFlow link. It is therefore also not possible for a remote node to traverse a linked list, and make edits to the linked lists, thus also not making it possible to use the malloc libraries on this linked list.

Another problem is preventing multiple nodes from accessing the linked list structure at the same time. While one node is for example defragmenting the linked list, another should not be able to allocate new objects in it. When programming on the same machine, developers usually employ locking mechanisms such as mutexes, or lock bits. These features are however not available as for the mutex case there is no single OS to keep track of mutexes, and for the lock bit case there are, again, the ThymesisFlow cache coherency issues.

The solution chosen to allow all of this is a centralized architecture. Every node in the cluster is responsible for managing its own memory region allocations. Only the node which owns the memory, will call malloc free and realloc on the memory regions. This architecture ensures no race conditions may occur, as all requests flow through the local node, which can employ local mutexes. Secondly, only the owning nodes caches need to be kept coherent. This is the supposed behavior of the CPU caches, only the local CPU writes to its own local memory. The linked list is therefore always kept coherent within the node which accesses the memory. No other node will access the linked list, and will therefore have no coherency issues.

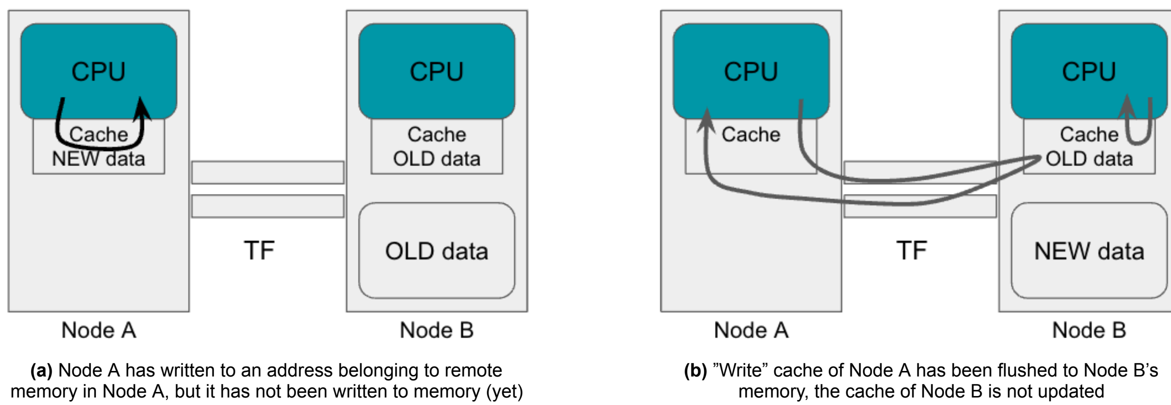
An edge-case is reallocation of memory. When an application calls realloc, it requests a memory region to be expanded to a certain size. In the easiest case there is still room after the memory region to be expanded into. A more difficult case occurs if there is no room. A new region with the newly requested size will to be allocated, and the data from the smaller region needs to be copied to this new region. When a remote node wants to reallocate a memory region, it can either do the malloc, copy, free operations itself, or it passes a realloc request to the memory owning node which does these operations locally. The second option was chosen because this would mean a potential copy operation does not need to go over the slower ThymesisFlow memory bus, but rather on the local faster memory bus. A remote node is able to call a realloc gRPC method on a memory owning node, where the memory owning node will return the pointer to the newly, potentially moved, memory region to be used by the caller.

3.8.2. Remote Arrow MemoryPool

As discussed in section 3.4 Arrow uses MemoryPool abstractions to decide which allocators to use, and to keep track of allocation statistics. To allow for easy developer experience a new Arrow MemoryPool has been created which transparently handles the discussed cache coherency flushing operations, remote malloc requests and allocations into these regions.

During initialization of the disaggregation API, RemoteMemoryPool class instances are instantiated. When an application wants to write a buffer into remote memory, it simple passes the corresponding RemoteMemoryPool into the existing Arrow AllocateBuffer, or Builder methods to use the remote memory. Below is a simple program which shows the ease of use for both methods.

```
1 // Get the device (node) we want to write to
2 ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::tf_device> dev, orc.GetDevice(1));
3 // A pointer of the RemoteMemoryPool is stored inside the device struct
4 std::shared_ptr<arrow::MemoryPool> memory_pool = dev->memory_pool;
5
6 // Create an unsigned int 16 builder which uses the RemoteMemoryPool,
7 //   which redirects allocations into remote memory
8 std::unique_ptr<arrow::UInt16Builder> uint16builderRemote =
9     std::make_unique<arrow::UInt16Builder>(memory_pool.get());
```



(a) Node A has written to an address belonging to remote memory in Node A, but it has not been written to memory (yet)

(b) "Write" cache of Node A has been flushed to Node B's memory, the cache of Node B is not updated

Figure 3.6: Write caches during remote writes.

```
10
11 // OR we create Buffer we write to ourselves
12 ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::Buffer> buff,
13     AllocateBuffer(SIZE, memory_pool.get()));
```

3.9. Cache coherency in remote memory when writing and reading

Now that allocations have been organized, an analysis is given into how to access the memory allocated. The goal is to have the node which writes into remote memory be cache coherent, as well as the node which owns the memory, and finally any other nodes which are also remote to the lended memory. As stated before, there are in this situation many CPU caches active. Namely:

1. CPU cache of the node which "owns" the memory
2. CPU cache of the node writing to the lender's memory. This may also be the lender itself.
3. CPU cache of any other node, remote to the "owning" node

In the following sections it is explained how every one of these three classes of CPU cache is kept coherent.

3.9.1. Reading from memory which was written to by its "owner"

The first case analyzed is when a node accesses its own memory. This is the most simple case, as any remote reads that come in, will simply snoop the local CPU cache before retrieving actual memory. When a write is pending in the CPU cache, a remote read will fetch the updated value from the CPU cache first.

However when the lender writes into its own memory, any CPU caches of borrowers are *not* updated. Thus, before a lender can write into its own memory, all remote nodes will have to flush their CPU caches of the memory region to be written to.

3.9.2. Remote writes: flushing local writes to remote memory from local CPU cache to remote memory

Just as described in subsection 3.7.1, there not only is an issue reading from remote memory, but also when writing to remote memory. When writing to remote memory addresses, the CPU updates its local cache and does not guarantee the data is actually written through to the remote memory through the ThymesisFlow link. This is illustrated in Figure 3.6a. When a node reads from remote memory, the correct data is returned, but is not fetched from ThymesisFlow, it is possibly returned from local cache. We therefore need to flush any memory we have written to remote ThymesisFlow addresses, to ensure they are actually written to remote memory.

3.9.3. Remote writes: invalidating remote cpu cache

After the flush to remote memory, the data is written to memory on the remote side. What has not been updated is the CPU cache on the remote memory. See Figure 3.6b. If the CPU of the owning memory has any cachelines stored of the written memory, any reads to the memory will fetch the cached values instead of the newly written data. CPU reads of the owning node will first fetch its cachelines, but also any remote ThymesisFlow/OpenCAPI transaction coming in will first snoop the CPU cache before fetching memory. The memory stored would never be reached. A possible solution would be to invalidate the caches of the CPU of the owning node, but as stated before (subsection 3.5.2), cache invalidation is not possible, we can only flush caches.

All caches need to be flushed before the data is written. We need to flush the caches of the memory owning node, but also the caches of any and all nodes wanting to use this data later on. This so that any reads following the creation of the table repopulates the CPU caches to the newly created data. This gives rise to the following workflow to write data to a remote region, to be later read by both lender, and any other remote nodes:

1. Allocate a memory in a remotely accessible memory region
2. Tell all nodes to flush their caches of this memory region. Both the lender, and any other borrower
3. Write data to the memory region
4. To ensure data is not hanging in the CPU cache of the writing node, flush the full memory region through the ThymesisFlow link to remote memory.

Note that the last step is not required when writing to shared *local* memory. As any transaction coming in from remote nodes will go though the OpenCAPI link, the local CPU cache will first be snooped before the local memory is loaded. In this case it does not matter if written memory is pending in the CPU cache, as any remote read, or local reads will first read the CPU cache.

3.9.4. Flushing is only required during initialization of Arrow objects

To summarize there are two flushing stages to do. First the remote memory addresses on all nodes are flushed, to ensure no caches will write old data into memory. Write the data, after which we force flush all written data to the remote memory. Now, when any node reads the created Arrow object, the data will be cached in its local CPU cache. The CPU cache will now have an up to date version.

An important thing to note here is that the flushing of caches is only done during the initialization of Arrow objects. After the initialization, Arrow objects are fully immutable, and we do not need to worry about cache coherency anymore as the underlying memory will not change. Any CPU caches will be usable, and will contain up to date data. Caches would therefore never have to be updated as the underlying memory never changes. The overhead of creating coherent Arrow objects is therefore present only at the initialization of the data.

3.10. Lifetime of a remote object

As discussed in subsection 2.1.6 Apache Arrow makes extensive use of the C++ *shared_ptr* object. These pointers are smart in the sense they take care of memory delete operations when all pointers have been overwritten or have gone out of scope. When we allocate objects in remote memory regions we need to decide who is responsible for deleting objects, and which node will free the underlying memory corresponding to objects.

We can differentiate three types of Arrow objects which can exist in the memory of an application:

1. A local Arrow object, instantiated by this node, and residing in local memory.
2. A local Arrow object, instantiated by another node, and residing in local memory.
3. A remote Arrow object, instantiated by this node, in the memory of another node.
4. A remote Arrow object, instantiated by another node, and in the memory of another node.

As we have discussed in subsection 3.8.1 every node is responsible for the memory management inside that node. The question here is which node in every case has the final responsibility of deleting Arrow objects. When working with purely local, non disaggregated, objects, the object is deleted when all references to the shared_ptr have gone out of scope or are deleted. This is done by the shared_ptr library by keeping track of how many instances of the shared_ptr still exist. In a disaggregated setup there is no possibility to keep track of the number or references in scope, as there is no coherent way to communicate counter values. We therefore choose to have the node which created the original Arrow object to also be responsible for free'ing the memory associated with that Arrow object. For all three cases this is solved by having the node which instantiated the original object, be responsible for calling free on the memory pointer.

For all four cases this gives a clear division who is responsible for deleting the pointer. More practically, the Arrow code in the 1st and 3rd case is responsible for calling free. When all references of the object described in case 2 and 4 go out of scope, the library should be no-op on the memory operation.

3.11. Overview: how an Arrow object is initialized

To summarize, an overview of the creation of a disaggregated Arrow object is given:

1. Allocate a memory region. If it is remote, call the gRPC method. If it is local call the relevant malloc call directly.
2. Tell all nodes to flush their caches of the allocated regions.
3. Write to the memory region.
4. If the memory written to is non-local. Flush the region to ensure no writes are held in CPU cache.
5. Finalize the disaggregated Arrow object, marking it immutable. All CPU caches do not contain the object, they will be repopulated with the newly written data.
6. Serialize the table descriptor of the Arrow object
7. Send the table descriptor to every node. Potentially every node has initialized their own part of the HugeTable. Thus every node will receive the table descriptor of every node.
8. Every node now combines the received table descriptors into a single HugeTable instance.

4 Implementation

This chapter will describe implementation details on how the systems described in the Methodology section are implemented in code. Modifications to Arrow will be made, and code is written to ensure the ThymesisFlow flushing operations.

4.1. Mallocs in mapped regions

For the purposes of this thesis a simple allocator written by Embedded Artistry LLC was used to manage the allocations inside the ThymesisFlow mapped memory regions. [21] The allocator makes use of linked list type memory regions, and has the ability to merge previously freed regions into a bigger contiguous chunks again to prevent fragmentation.

The code allows for multiple regions to be defined, every region can have their own malloc structure which keeps track of allocations.

4.2. CMAKE mods

For a program to compile, often cmake is used to configure and check the build environment. Cmake generates a build environment where for example the system libraries are linked to, and cmake generates the compile commands.

Without cmake a lot of configuration is left up to the developer. For example cmake is able to select the correct version of a library, is able to generate compiler configs for both dynamic and static linking, and is even able to install libraries from GitHub and compile them. The main output of cmake is a build environment where the *make* command can be run inside in. When make is called, the program is compiled in accordance with the specifications defined in the cmake configs. *make* in its turn calls the system compiler binaries.

Thus if we want to modify the build environment of the Arrow project, we change the cmake configs. The cmake config will then generate modified make files, and make will be able to call the compiler with correct parameters, and correct source file dependency order. Arrow has a very complex cmake structure. Arrow has many different extensions, all of which can be chosen to be compiled or not. A developer can for example choose to omit the CSV loading functionality from the Arrow binary to allow for a smaller binary. Or load in the compute kernel functionality to allow for execution graphs on Arrow data.

The work in this thesis consists of adding new source files, but also to link in the gRPC library. Options for cmake were added to allow for the ThymesisFlow extensions to be chosen to be compiled in the Arrow library. When the *ARROW_THYMESISFLOW* option is set to ON, Arrow will compile all relevant source files, install gRPC locally and allow for compiling the Orchestrator logic. Furthermore during compilation a step is added to compile the protobuf definitions into C++ source files.

4.2.1. cmake config for compiling protobuf definitions

Enabling the *ARROW_THYMESISFLOW* option also sets gRPC and Protobuf as a dependency. When cmake is run, these libraries will be downloaded from GitHub, and compiled from source.

As described in section 3.1, Protobuf allows for a unified serialization format for in-transport communication. A user application can define a message in the Proto language, and protobuf compiles the definition file into C++ source code which serializes data to the specified packet. During the compile step of the Apache Arrow library, this Protobuf compiler also needs to be called.

This is done by configuring cmake to first install gRPC and protobuf, compiling and linking these libraries to each other, after which the protobuf compiler is available. cmake then will compile the protobuf definitions into C++ files in the cmake build directory. The generated files are called:
`tf_orchestrate.grpc.<grpc | pb>.<cc | h>`
From the Arrow source we can then simply include them with
`#include "tf_orchestrate.pb.h"` and `#include "tf_orchestrate.grpc.h"`.

4.3. gRPC

With the protobuf and gRPC code snippets generated, and the libraries linked into the Arrow libraries, we can use them in development. gRPC works in a server-client model. A server has a service derived from the protobuf definitions of which the behavior needs to be defined. On the client the same happens, the service derived from the protobuf needs to be filled in by the developer.

A simple example is the malloc service. A client can use this service to request a memory region inside the memory region of the server. On the client side we simply want to call `void * malloc(size)`, and the gRPC and protobuf code should handle serializing the message with the size, call the server side service, and deserialize the response. The c++ code will block, and finally return the pointer to the allocated region. The client gRPC therefore will look something like this:

```
Result<uintptr_t *> OrchestratorClient::Malloc(uint32_t size) {
  grpc_tfo::MallocRequest req;
  grpc_tfo::MallocReply reply;
  grpc::ClientContext context;
  grpc::Status status;

  req.set_size(size);

  status = stub_->Malloc(&context, req, &reply);
  if (!status.ok()) {
    std::stringstream ss;
    return Status::IOError(ss.str());
  }

  return (uintptr_t *)reply.address();
}
```

The power of gRPC becomes really clear when the application wants to execute this code. It simply calls:

```
  ARROW_ASSIGN_OR_RAISE(uintptr_t* out_p;, client->Malloc(size));
```

And the `out_p` variable contains a pointer retrieved from a remote server. gRPC will instantiate a connection, will handle timeouts, and will handle serialization. The application only needs to do a method call.

The server code has a similar setup. A method is extended form the gRPC generated files, and looks like the below code. gRPC will handle starting thread pools which handle incoming RPC requests, and will abstract away error handling, session handling and security.

```
class OrchestratorServerImpl final : public grpc_tfo::OrchestratorServer::Service {
  grpc::Status Malloc(grpc::ServerContext* context, const grpc_tfo::MallocRequest* request,
                             grpc_tfo::MallocReply* reply) override {

```

```
    MemoryPool* pool = default_memory_pool();
    uint8_t* address;
    Status status = pool->Allocate(request->size(), &address);
    if (!status.ok()) {
      reply->set_address(0);
      return grpc::Status::OK;
    }

    reply->set_address((uint64_t)address);
    return grpc::Status::OK;
  }

  ...
}
```

As the gRPC server consists of many threads, mutexes are used to prevent race conditions. All methods which are called from the gRPC server services is protected with mutex lock_guards. This ensures only one allocation can happen at the same time.

4.4. Local MemoryPool and proxied remote MemoryPool

As discussed in section 3.4 Arrow calls the MemoryPool class methods when an object needs memory allocated. The MemoryPool class is the perfect place to insert the proxy to remote memory requests. When an application calls an Arrow method which may require allocations, it allows for the application to pass which MemoryPool the allocations should be done with.

This thesis has taken the route where a node has a MemoryPool instance for every remote region it can allocate into. This is called a *RemoteMemoryPool*. A RemoteMemoryPool proxies all malloc/free/realloc requests to the relevant node which the memory belongs to via gRPC. The RemoteMemoryPool->Allocate() method for example will call the aforementioned Malloc gRPC call.

For example when an application wants to write data into an array using an Arrow Buffer builder into remote memory, it will simply call:

```
// Retrieve device config
ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::tf_device> dev, orc.GetDevice(1));
// Extract memory_pool instance of this nodes memory
auto remote_memory_pool = dev->memory_pool.get();
// Instantiate an Arrow builder, telling it to use the remote MemoryPool for allocations
std::unique_ptr<arrow::UInt16Builder> remote_builder = std::make_unique<arrow::UInt16Builder
    >(remote_memory_pool.get());
// Create local data
uint16_t local_array[SIZE*2] = ...;
// Append local data to builder. This will copy data into remote memory. Will also realloc
    when needed.
ARROW_RETURN_NOT_OK(remote_builder->AppendValues(local_array, SIZE));
// Finalize instatiation. Create an immutable Arrow Array object.
ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::Array> remote_array, remote_builder->Finish());
```

This is all the user application has to do to make a cache coherent disaggregated Arrow object in remote memory! The underlying system will execute the pre-write flushes, will communicate malloc requests to remote, and will flush any written data when the builder is finished into a finalized Arrow object.

4.5. Serializer

As stated before serialization code from the Fletcher [19] project was modified to allow for serializing RecordBatches into table descriptors. The code was modified to support some extra data types, and now also serializes into protobuf messages to be transmitted over the gRPC connections.

As supporting any and all data types Arrow has defined is out of scope for this thesis, the serializer is only able to handle fixed-width data types. The RecordBatch table descriptor is serialized into the following protobuf message:

```
message TableDescriptor {
  message Field {
    message Buffer {
      // As we will not be writing from remote when the
      // object is finalized, this is not necessary for
      // the other party to know
      // uint32 capacity = 1;
      int64 size = 2;
      uint64 pointer = 3;
      uint32 level = 4;
    }

    bytes name = 1;
    uint64 type = 2;
    int64 length = 3;
    int64 null_count = 4;
    repeated Buffer buffer = 5;
  }

  uint64 rows = 1;
  uint64 start_index = 2;
  repeated Field field = 3;
}
```

4.6. Synchronization bit waits

A synchronization system was developed which allows different nodes to wait for each other to reach a certain program state. This is required when a node for example should wait on another node to send it a table. Or it is required during application startup to wait for all nodes to become ready.

The system works with state flag vectors, where every bit has either a 1 or 0. Every node has its own single state vector which it can set locally. This vector is used throughout program execution and initialized to zero at startup. A node is able to wait for all nodes in a cluster to reach a certain state, by waiting for a specific flag to become one or zero.

When a node reaches a certain state, it will set its state vector to the updated state. In most test benchmarks written for this thesis for example, bit 0 set to 1, signals that the node has completed all initialization steps. After a bit is set, usually the node needs to wait for all other nodes to have the same bit set. This is done using the *waitFor* method. A node passes a flag bit vector and a mask for this vector. The flag bit vector tells the wait function what values to wait for, the mask tells the function which bits are relevant. How this practically works is that the *waitFor* function serially checks every nodes flag state vector, if it is not yet what we expect, it blocks until it is.

To check if a node has reached a certain state a gRPC request is executed to fetch its current state vector. If the vector is not yet what we expect, the vector is fetched every 0.5ms until it is. These checks do not impact the performance of the other node as gRPC starts thread pools which handle the requests.

4.7. HugeTable logic

As stated in section 3.6, the system is able to instantiate tables which span the memory of multiple nodes. The procedure for creating such a table has been chosen to be as follows:

1. First every node initializes the local memory parts, and creates RecordBatches from this data. This creation is done using the disaggregation flushing steps described before. In the current state every node has a locally stored remotely accessible RecordBatch, but the other nodes do not know yet of its existence.
2. Next every node broadcasts the table descriptor of the RecordBatch to all other nodes, combined with an index where in the HugeTable it belongs. For example a node may instantiate a Record-Batch which contains rows 1000 to 2000 of the HugeTable, it will broadcast this with index 1000.

3. Every node will receive the table descriptor broadcasts with corresponding indexes, and store them in an array.
4. Finally every node will combine all the received RecordBatch table descriptors into a HugeTable. The list of table descriptors is sorted by index, checks are done to see if the indexes and row counts form a contiguous unit.

Any application code which supports Arrow Tables, will now be able to work on this disaggregated HugeTable object. For example using the Arrow compute kernels, we can sum together all values of a column, meaning we sum the values stored in every node. Under the hood the compute kernel will loop through all the arrays stored in the Huge Table, and simply access their addresses. If the address is remote, ThymesisFlow will redirect the memory access to the remote memory.

4.8. Type of flush instructions

The flushing instruction used to flush cache lines to memory is the *dcbf* instruction. This instruction however, has 3 parameters. *RA*, *RB*, and L. [20, p. 1064]

With the RA and RB parameters we can influence which cache blocks are flushed. For the simple use case of flushing a cache line at a certain address RA will be set to 0, RB will then be equivalent to the EA to be flushed. As per the documentation:

> The dcbf instruction calculates an effective address (EA) by adding the contents of general-purpose register (GPR) RA to the contents of GPR RB. If the RA field is 0, EA is the sum of the contents of RB and 0.

With the L parameter the application can influence which caches are flushed. The most broad flush is the L=0 value. L=0 says any writes in the data cache are to be flushed, and the data block to be invalidated in all processors in this coherence domain. L=1 is more limiting in that it only flushes the pending write, but does not invalidate the cache block. L=3 is for only flushing the primary data cache. L=4,6 are for when the EA referenced has underlying persistent storage, which is not the case for RAM. L=0 is the only one which fulfills the use case of invalidating and flushing the write caches, thus that is the one used for the coherency protocols.

4.9. Memory barrier instructions

Before and after the sync instructions, memory barriers are placed. As described in subsection 3.5.3 these instructions ensure memory instructions before the flushing to be finished, and memory instructions after the flushes to execute after flushes. This concern comes from the fact that the processor may re-order instructions before and after the flushing instructions, due to them not having a clear derivable dependency.

A generic sync instruction for in the Power architecture is `sync L,SC`. [20, p. 1086] SC influences how the processor re-orders store instructions. A non-zero SC means that only store instructions are prevented to be moved across the barrier. The L parameter influences which instructions are prevented from being re-ordered. An L=0 means no other instructions will be allowed to cross the barrier, while L=1 will only prevent Load, Store and dcbz instructions to cross. L=0 is also called a *heavyweight sync*, meaning it is the most aggressive memory barrier.

For the purposes of this thesis the heavyweight sync instruction is used, as the overhead from causing a single pipeline bubble before and after flushing operations is negligible.

4.10. Deletion of objects

In section 3.10 four cases were given how an Arrow object can exist in the memory of a local object. The main design decision chosen was to have the node which has initiated a malloc call, also be the one who has to call free on that object.

This has to hold for pure objects, which have been fully allocated by a single node, but also for compound objects such as the HugeTable as described before. The HugeTable may have many buffers which are allocated by different nodes in different memory regions. When the last shared_ptr of an Arrow object goes out of scope, the C++ compiler will call a recursive delete operation on all sub classes instantiated under it until it finally calls delete on the PoolBuffer class. The PoolBuffer class in Arrow keeps track of in which MemoryPool a Buffer object was created in. When the PoolBuffer is called delete on, it will call the corresponding MemoryPool::Free method.

In the case of a locally allocated object the PoolBuffer will point to a local MemoryPool instance, calling free on it directly. When delete is called on a disaggregated object, PoolBuffer will point to the RemoteMemoryPool class, as created and described in this thesis, which has a different behavior. The RemoteMemoryPool has a local list of addresses which it has malloc'ed, and will only call free if the memory to be freeed, is also originally malloc'ed by this node. This ensures that in the case of a HugeTable going out of scope, Arrow will iterate through every buffer contained in the HugeTable, and only free the memory the node has allocated locally.

4.11. Orchestrator: User facing API

All features mentioned in this thesis are controllable by the user application in the API described in this section. The API used to control these systems is called the *Orchestrator*. The orchestrator API allows for defining the cluster setup, handles mapping ThymesisFlow memory regions, and handles configuring the Arrow objects. A list of the methods callable by the application:

- *AddDevice and derivatives*: For defining the general structure of the cluster. We call this for every local memory region, and for every remote memory device. Parameters are: IP address to connect to, start of the memory region of this device as mapped into memory, and the size of the memory chunk.
- *MapDevice*: Map the given memory device into the current program's memory. The mapping location is fixed using the mmap MAP_FIXED flag (section 3.3) to the location defined by the AddDevice parameter
- *InitializeServer*: Initializes gRPC server so that other nodes can pass messages to this node
- *WaitAllDevicesOnline*: Blocking wait until all devices are also waiting for all devices to come online
- *SendRecordBatchMD*: Sends a given RecordBatch to every node *without* the data, only the table descriptor.
- *SendTableFull*: Sends a full table including the data buffers content.
- *DebugRecordBatchPrintMetadata*: Prints all buffer pointers contained in the RecordBatch.
- *GetHugeTable*: Gathers all received RecordBatches from remote nodes and the locally added RecordBatches, and combines them into a single HugeTable.
- *SyncWait*: Sets own state to specified state, and blockingly waits for all other nodes to reach same state.

4.12. Security

A noteworthy part which has been completely omitted from this thesis is the security aspect of the whole system. As this thesis has developed a proof of concept, the design of security features is left to be developed for a production ready design.

For an outsider node, without being part of the ThymesisFlow network, only the gRPC link is accessible. Requiring authentication on the gRPC server will at least limit the attack surface to applications already running inside the ThymesisFlow network.

The following features should at least be implemented to increase security:

- The gRPC link is accessible without credentials.

- HugeTable chunks received are not checked for validity. The buffer pointers contained within should be checked if they reside in the a memory region which is shared.
- The free gRPC service has no checks to see if the free is allowed for that address. A node is able to free memory which it has not allocated.
- Rate limiting and parameter limits on memory intensive gRPC calls. It is now possible to crash a node using a single big Malloc call.
- For the ThymesisFlow setup by HPI both nodes have a memory device located at `/dev/mishmem-s1`, which is read and writable by every user on the system. As these devices will contain possibly sensitive Arrow data, they can be secured with simple Unix permissions.

4.13. Test setup

Several setups have been used throughout this thesis to verify the correct operation of code written. Initially the test setup as provided by HPI was not yet available, so local test setups were devised to allow for functional testing.

4.13.1. Shared memory mappings

The simplest form of a test setup was interchanging data between two processes on the same machine by using a Linux shared memory mapping.

This testbed was able to be run on a single machine, where every node is a separate process. As ThymesisFlow is essentially mapping a remote piece of memory into local userspace memory, we can also achieve this by using the Linux Shared Memory devices. These devices appear as files in the filesystem, but are actually memory only files. Mapping these memory regions into userspace memory can be done using the *mmap* syscall. These special shared memory devices also allow for multiple processes to map the same shared memory device, thus allowing for a shared memory region between to processes. We also use the MAP_FIXED mmap flag to tell the kernel the address we pass, is an exact address. The memory device is to be mapped at the exact location specified.

The shared memory mapping is almost equivalent to the ThymesisFlow setup. The memory written to the device, is immediately visible at the other process. It also is mmap'able, just as the ThymesisFlow regions are. But, the cache coherency issues are not at all present in the shared memory devices. As both processes are in the same NUMA domain, they are fully cache coherent with each other, and thus the memory region is also cache coherent. Finally, as is to be expected, the latency and memory throughput is not simulated using this setup. The memory access times from both processes is identical, and they are near instantaneous as memory accesses can stay within the cache of the processor.

This setup was tested on both a desktop Ubuntu system, as well as a MacBook Pro with an M1 chip. With some minor modifications to the build system of Apache Arrow, it was able to compile and work successfully on the M1 `AArch64` architecture.

4.13.2. QEMU: Shared virtual PCIe memory device

To ensure no accidental use of any kernel features which will not be available on the real ThymesisFlow system, a test was run with different virtual machines. Every node gets its own QEMU virtual machine, and ThymesisFlow memory is simulated using virtual PCIe memory cards. Two memory cards are created, and they are attached to both virtual machines. Each PCIe device will "belong" to one of the virtual machines. Note that in the case of QEMU PCIe memory mappings, these are also kept cache coherent by the QEMU emulator. Thus, also here we cannot verify our assumptions about ThymesisFlow, OpenCAPI and Power 9 cache behavior. We can verify that no local OS features are used.

The QEMU attached PCIe device is supported by the Linux kernel, and creates a memory device in `/sys/bus/pci/devices/0000:PCI_BUS:PCI_SLOT.PCI_FUNCTION/resourceBAR`. In the case of the QEMU device this results in a memory device at `/sys/bus/pci/devices/0000:07:01.0/resource2`

where the bus slot, and function are defined in the QEMU config. These devices can be mapped into an application memory space using regular mmap syscalls.

The basic functionality was verified. Arrow is able to allocate locally inside the mapped regions, is able to allocate on remote regions, and all synchronization and network communication using gRPC is working.

4.13.3. ThymesisFlow setup by Hasso Plattner Institute

When all code was finished and tested on the simulated setups, we moved to executing the system on a real ThymesisFlow system. The Hasso Plattner Institute, or HPI for short, has a two system ThymesisFlow setup in their lab. Both these systems have dual socket Power 9 processors in them, 512GB of RAM, and both of them have an Alphadata 9V3 FPGA installed. More details of the machine specifications in section 5.1.

Limitation of only one ThymesisFlow link

Important to note here is that these nodes have only one ThymesisFlow link between them. In ThymesisFlow a node is either a lender, or a borrower of memory. A node cannot be both. This is a limitation in both the Linux kernel driver of ThymesisFlow, as also the FPGA design. This means we can only share the memory of a single node, the other node can only access its own local memory, and the remote memory. For a fully disaggregated system where every node can access all other nodes' memory, we would need N^2 number of links in between them. Measurements and experiments in this thesis focus on the case where only the borrower node can access the memory of the lender, and the lender only being able to access local memory.

thymesisflow-agent

HPI has developed a so-called `thymesisflow-agent`. This software agent allows for multiple users of the system to use the ThymesisFlow stack. The agent configures the OpenCAPI link and configures ThymesisFlow. Finally the agent provides a shared memory device at `/dev/mishmem-s1`, which is available on both the lender and the borrower.

These memory files make developing for ThymesisFlow considerably easier. If a user application wants to share memory, it can simply call the *mmap* syscall on both the lender and borrower on this `/dev/mishmem-s1` file. A program which uses ThymesisFlow therefore does not need link any ThymesisFlow or OpenCAPI libraries, and does not need to concern itself with low-level hardware initialization. This results in a mapped memory region in both applications, belonging to the lender. If the borrower writes to this region, ThymesisFlow will bridge these transactions to the OpenCAPI link of the lender. If the lender writes to this memory region it will simply pass into local memory.

Bring-up of the test setup

The bitfile of the FPGA's is handled during startup of the machine. A short manual on how to initialize the `/dev/mishmem-s1` mappings can be found in Appendix B.

5 Results

The proposed system is one in which ThymesisFlow and Apache Arrow are combined and modified into a fully cache coherent system. The power of the proposed solution is that the overhead of making the system coherent, is fully located in the initialization of the data. An application will only incur performance hits in the creation of an Arrow data objects, any reads afterwards are performance limited by the ThymesisFlow system. For an application developer to consider the system proposed it is not only important to measure the performance hits caused by initialization, but also analyze what the latency and maximum bandwidth is of the system. The ThymesisFlow creators have outlined some performance metrics regarding the latency and STREAM benchmark times [1, 22], what is missing is an analysis on what access patterns work well on the system.

To allow an application developer to chose the Arrow ThymesisFlow combination it is important to know what overheads exist in the system. More specifically measurements are given on:

- What the performance is of the initialization of a disaggregated Arrow object. Individual components of the coherency protocols are split into smaller measurements.
- Comparison of the copying of a table across an ethernet link, to fully accessing a table through ThymesisFlow.
- Read and write performance benchmarks for varying thread numbers, different type bit widths, serial access, and finally, strided access patterns.

Every benchmark is guaranteed to be memory limited, as they have been analyzed with the perf tools. This ensures memory stalls happen on the memory access instructions, and the benchmarks are not compute limited.

5.1. Test setup

All experiments are run on the ThymesisFlow test setup of Hasso Plattner Institute. This setup consists of two *IBM Power9 IC922* servers. These servers both have a OpenCAPI enabled Alpha Data 9V3 FPGA, with glass fiber connections in-between. These FPGAs have the ThymesisFlow firmware flashed on them. The firmware is responsible for translating processor bus addresses into effective addresses on the memory lending side. Both servers have two Power9 CPUs in their sockets. The CPUs each have the following specifications:

- Architecture ppc64le
- 2 out of 4 CPU sockets populated
- Every CPU has 12 cores, 4 threads per core, in SMT-4 configuration.
- 768KiB L1d cache, 768KiB L1i cache, 6MiB L2 cache, 120 MiB L3 cache
- Linux ic922 5.4.0-120-generic SMP enabled kernel

- Ubuntu Focal 20.04.4 LTS
- Memory: 16 x ECC DDR4 2666MHz 32GiB. 8 banks per CPU socket
- Network card: MT27710 [ConnectX-4 Lx]: 25Gbit/s

The tests were done on these two machines. Note that there is only a one-way link in this setup. Node 03 is always a borrower node, and is not able to borrow its memory to the 04 node.

The cache sizes are relevant to keep in mind when deciding data sizes for memory bandwidth measurements. The data sizes written and read from memory need to be bigger than the available cache to ensure that data is actually written and read from memory.

5.1.1. NUMA domains

Every server has 2 CPUs connected, and each CPUs has its own 8 memory banks. Important to note here is that Linux decides for every process where the memory is stored, and in which CPU core the process is run on. These systems are called the NUMA extensions in Linux. In general Linux will allocate memory which has the *closest distance* to the CPU core the process is running on. An application does have some influence on where processes are run on using the `numactl` utility. The output of `numactl -H` for example shows the context in which the system operates:

```
1  philip.groet@ic922-04:~$ numactl -H
2  available: 4 nodes (0,8,48,64)
3  node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
       30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
4  node 0 size: 261683 MB
5  node 0 free: 250728 MB
6  node 8 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
       75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
7  node 8 size: 260735 MB
8  node 8 free: 216951 MB
9  node 48 cpus:
10 node 48 size: 0 MB
11 node 48 free: 0 MB
12 node 64 cpus:
13 node 64 size: 0 MB
14 node 64 free: 0 MB
15 node distances:
16 node   0   8  48  64
17   0:  10  40  80  80
18   8:  40  10  80  80
19  48:  80  80  10  80
20  64:  80  80  80  10
```

Here we can see to which CPU cores every NUMA domain belongs. In this case, every CPU socket has one NUMA domain. We can also see how much memory is attached to each. Finally at the bottom we can see the "distance weights", which the NUMA extensions use to optimize locality between memory and process core location.

For every experiment in this chapter we force the experiments to run on NUMA node 0, and force the memory to be stored on memory banks belonging to NUMA node 0. Node 0 was chosen as the FPGA OpenCAPI cards are attached to this NUMA domain. This is done using:
`numactl -N 0 -m 0 <COMMAND>`

5.2. Linux perf tools to guarantee benchmarks are memory limited

To measure the maximum memory bandwidth possible, the application needs to be very memory intensive. The processor should dispatch memory requests faster than the underlying memory bus is able to handle them. In other words, the arithmic intensity needs to be very low, the ratio of compute and memory instructions should be low. When the memory bus of the processor is saturated, the internal processor pipelines will stall. Processors do not execute the instruction back to back, rather they are overlapped. This is called pipelining. While one instruction is loading, another instruction may for

Compiler optimization	Threads	Loop unrolling	SIMD	Bandwidth [GB/s]
-O0	1	N	N	1.24 GB/s
-O0	OMP dynamic, max(48)	N	N	0.57 GB/s
-O0	OMP dynamic, max(48)	Y	N	0.57 GB/s
-O0	OMP dynamic, max(48)	Y	Y	0.57 GB/s
-O2	1	N	N	1.76 GB/s
-O2	OMP dynamic, max(48)	N	N	13.96 GB/s
-O2	OMP dynamic, max(48)	Y	N	195.70 GB/s
-O2	OMP dynamic, max(48)	Y	Y	242.35 GB/s

Table 5.1: Effect of compiler optimizations and parallelism parameters on local memory write bandwidth

example be executing. When the underlying memory hardware is not able to keep up with the speed at which memory requests are dispatched from the pipeline, the pipeline is *stalled*. The program execution is halted, and has to wait for the stall to be resolved. The kernel may then switch to different threads as to fill in the stalled cycles with more instructions which do not stall.

We can use these stall events to measure how saturated the memory bus is. When most of the time is spent waiting for a stalled memory instruction to be executed, the memory bus is clearly saturated as it is not able to keep up with the workload. Linux has the `perf` tool set which is able to measure memory stalls occurrences, and localize which instructions caused them. It does so in a way where the program performance is minimally impacted. Perf is able to instrument CPU performance counters, and profile running processes. When a program is executed under the `perf record` environment, the program is randomly interrupted using for example hardware cycle counters, and the current state of the program is saved as ”events”.

For the purpose of testing if the memory bus was saturated by memory requests towards ThymesisFlow memory, the event called *stalled-cycles-backend* was used. This event signals the pipeline was waiting for underlying resources to become available, usually memory accesses. The `perf report` tool will then localize which instructions have most likely caused this event. For every experiment it was verified that the memory access instructions within the benchmarks were >95% of the time memory stalls.

An example *perf report* of a properly optimized for loop is given below. We see that 99.78 percent of the back-end stalls occur during the vector instruction `stxvd2x` which is a memory access instruction.

```
1   0.10         xxlor   vs0,vs32,vs32
2   0.08         vaddudm v0,v0,v1
3              for (uint64_t i = 0; i < SIZE; i++) {
4              years_buffer_data[i] = i;
5  99.78         stxvd2x vs0,0,r10
6   0.03         addi    r10,r10,16
7   0.01  →      bdnz    31c0 <RunMain(int, char**) [clone ._omp_fn.0]+0xc0>
```

5.3. Removing compute bottlenecks in benchmarks

Using the perf tools, initially the non-ThymesisFlow (local memory accesses) benchmarks ran at a mere 0.164GB/s. Obviously this is far away from the expected 120+ GB/s Power9 is able to achieve. This is on a program without compiler optimizations, single-threaded, no SIMD, and only one operation per write iteration.

In table Table 5.1 measurements are shown for differing parallelization setting for OpenMP, and compiler optimizations enabled and disabled. Notably we can see the Compiler optimizations to have a big effect. Looking at the instructions generated by the compiler, it is clear that without the optimizations, the compiler does not optimize indexing in loops at all. The index of the memory to write to is recalculated in every loop, and the memory address offset calculations is redone in every iteration. The perf tools confirm that the memory access instruction was only 12% of the time the reason for a back-end stall. In the last case with all optimizations and parallelizations enabled, the memory access instructions were 99.6% of the reason the program was stalled, correctly saturating the memory write queues.

5.4. Initializing an Arrow object

As discussed in section 3.11, creating a coherent disaggregated object requires several steps during initialization. To reiterate:

1. Call malloc, allocating a memory block for the data object. On either remote or local memory. If on remote, a gRPC call is done.
2. Tell all nodes to flush their CPU caches of the allocated region. This ensures no pending write overwrite the data, and no read caches are left anywhere.
3. Write the data to the memory address. Not Arrow specific, but ThymesisFlow specific. We will analyze this in detail in section 5.6.
4. If memory region is non-local: Flush written data from local CPU cache, to remote memory
5. Data is made to be immutable. Initialization is completed and an Arrow object is created.
6. Broadcast the created table descriptor to all nodes. All nodes now know where the data is stored and what type it is. Any reads from the data will repopulate CPU caches.

5.4.1. gRPC overhead

gRPC is used to facilitate communication between nodes. Measurement were taken how much time it takes for a simple gRPC call to execute by implementing an empty service which a client can call. This is essentially a ”ping pong” where the only code executed is the gRPC client and server code. An average of $3.23ms$ was measured over 10 iterations, $\sigma^2 = 0.636$.

Note that the gRPC C++ server implementation starts a thread pool of handlers which can each process incoming RPC calls. In the implementation created we employ locking mechanisms to communicate with the main application thread. These locks may incur additional delays. The end-to-end tests have all of these delays included.

5.4.2. Flushing overhead

Benchmarking flushing is less straight forward. Depending on how much data is in the cache, flushing will vary in time it takes. We therefore take several cases to give an idea how long it takes.

- *Local/Remote*: Flush data written to local memory addresses, but also flush data written to remote addresses which may be pending in local CPU cache.
- *Untouched / Touched memory*: Flushing of data which has not been touched, and flushing of data of which the application has touched every byte. Note that the CPU caches are smaller than the array touched, thus not the full array will be held in cache. As stated in section 5.1 and subsection 2.8.3 the cache sizes were described to be less than 150MiB total, while the array under test is 1GiB.

In Table 5.2 we see that when the cache is empty, flush times are faster than when the cache is full. Furthermore, ThymesisFlow bandwidth limit is not achieved as the remote and local flushing times are similar. For a remotely written data array, we do see a difference between remote and local, indicating ThymesisFlow is limiting.

	Time [ms]	
	$\bar{X}$	σ
empty, local	53.12	0.007
empty, remote	51.84	0.016
touched, local	60.32	1.71
touched, remote	88.57	1.53

Table 5.2: Time to flush 1GiB of written data. Approximately 8,388,608 flush instructions

	Time [ms]	
	$\bar{X}$	σ
Local	3.13	0.48
Remote	4.99	0.86

Table 5.4: Time for either a remote or local malloc call to complete

5.4.3. (De-)Serializing table descriptor

As discussed, serializing a table descriptor is quite fast. Only the column names and types, and the pointers to the arrays contained in the columns need to be serialized. 10 Iterations were done, the average is shown in Table 5.3.

	Time [ms]	
	$\bar{X}$	σ
Serialization	0.058	0.032
Deserialization	0.019	0.029

Table 5.3: Time to (de-)serialize a 3 column RecordBatch with column types: int64, int8. uint16

Serialization is done using the visitor pattern of Arrow. The visitor classes are very versatile, but incur performance penalties. For deserializing, a simpler approach was used where only fixed width data types are supported. This makes the deserialization faster than serialization.

5.4.4. Malloc

The malloc implementation created uses linked list data structures to reserve allocation space. See section 3.4. A benchmark was created which calls the malloc method on a local and ThymesisFlow remote memory region. Malloc is called to allocate a random region between 0 and 5000 bytes. Although this give varying sizes, a real Arrow object will call malloc with much bigger size requests. For the remote malloc implementation, the malloc request is communicated to the remote node through the gRPC link, malloc is called locally, and the resulting address is returned through gRPC. The results of these measurements are given in Table 5.4. The difference between remote and local is the gRPC call, which is similar to the time measured before.

5.5. Comparing to a full ethernet copy

We compare the transferring of a ”zero-copy, zero-serialization table” with a full copy over an ethernet connection. In the zero-copy case we only serialize the table descriptor, and we send this descriptor to the other node. In the full copy case we use the Arrow native table serialization calls. These native serialization calls include table descriptors, but also the data itself. This shows the power of the combination of Arrow and ThymesisFlow. With this combination we only need to send over the descriptor, after which data can be selectively accessed through the ThymesisFlow link. Therefore to get a realistic view an application developer should not only look at this comparison, but also take into account the access patterns necessary. See section 5.6 for an analysis on performance of access patterns through the ThymesisFlow link.

The table used is again a 3 column RecordBatch with $150 * 1024 * 1024 = 157.286.400$ rows. The data types of the columns are int64, int8, and uint16; bringing the total approximate table size to $11 * \texttt{Rows} = 1,61GiB$. For differing table slices we tested the transfer times in Figure 5.1.

5.6. ThymesisFlow micro-benchmarks

In this section we look purely at ThymesisFlow access times. Compared to local memory accesses ThymesisFlow has a slightly higher latency, but a much more limited memory bandwidth. Benchmarks

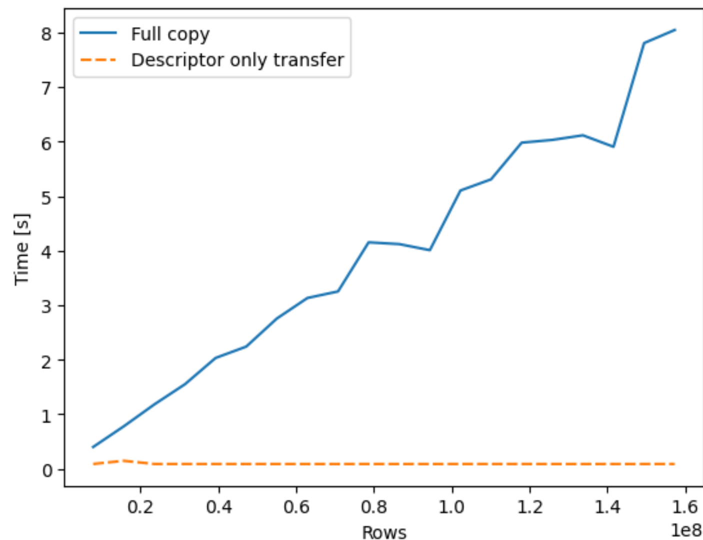


Figure 5.1: Transfer time comparison between full and descriptor only copy of an Arrow RecordBatch through the ethernet connection

have been done with varying thread counts, type bit widths and read or write patterns. An analysis was also done on how ThymesisFlow behaves under strided access patterns, as it is theorized these are the most difficult for ThymesisFlow to do.

5.6.1. Varying data types and thread count

To get an idea how the ThymesisFlow link operates under different conditions, tests were done under read and write conditions, local and remote memory accesses, narrow and wide data types, and different number of threads. The results are shown in Figure 5.2. The 4 threads number was chosen as to be comparable to the ThymesisFlow paper results. 48 Threads was chosen as that is the number of threads available for a single Power9 CPU.

Note though that in these measurements the number of threads was forced to be a certain number. By default OpenMP, the threading library used, takes the number of threads as an upper limit. OpenMP will dynamically scale the number of threads to prevent performance hits. As the number of threads was forced to a certain number, performance may in certain circumstances be better with dynamic thread counts.

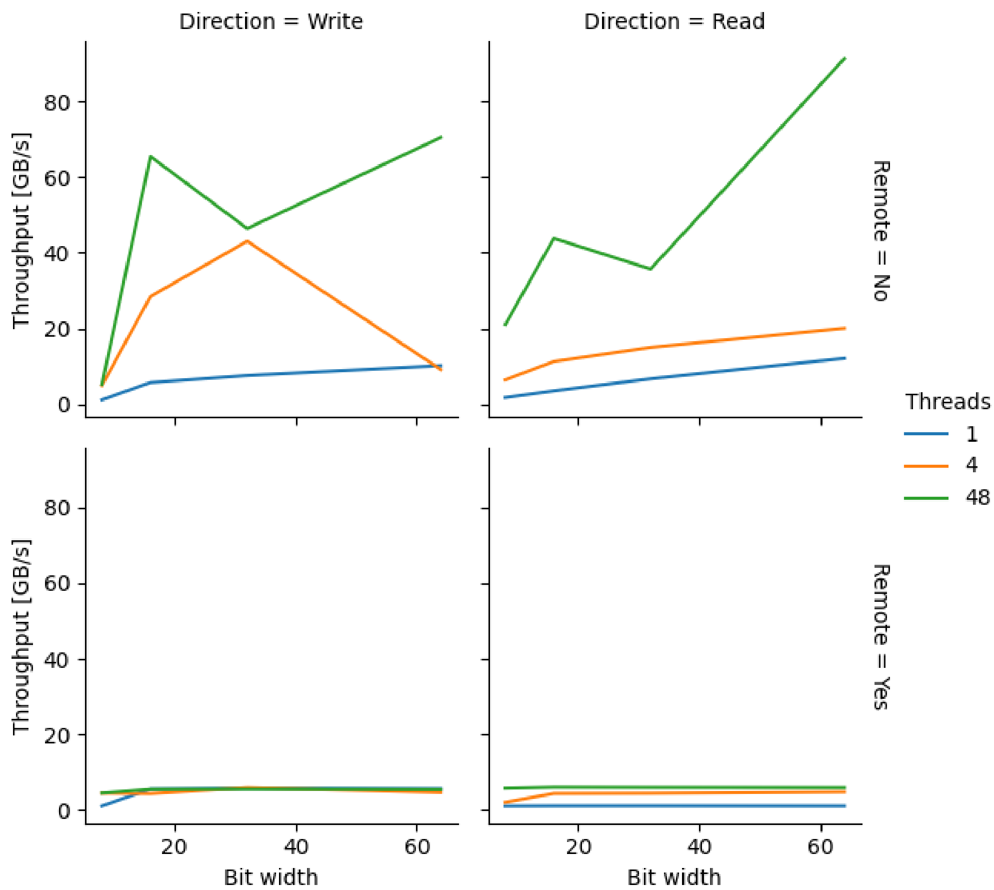


Figure 5.2: Max memory throughput for differing threads, Read/Write, ThymesisFlow remote/Local, and datatype bit width

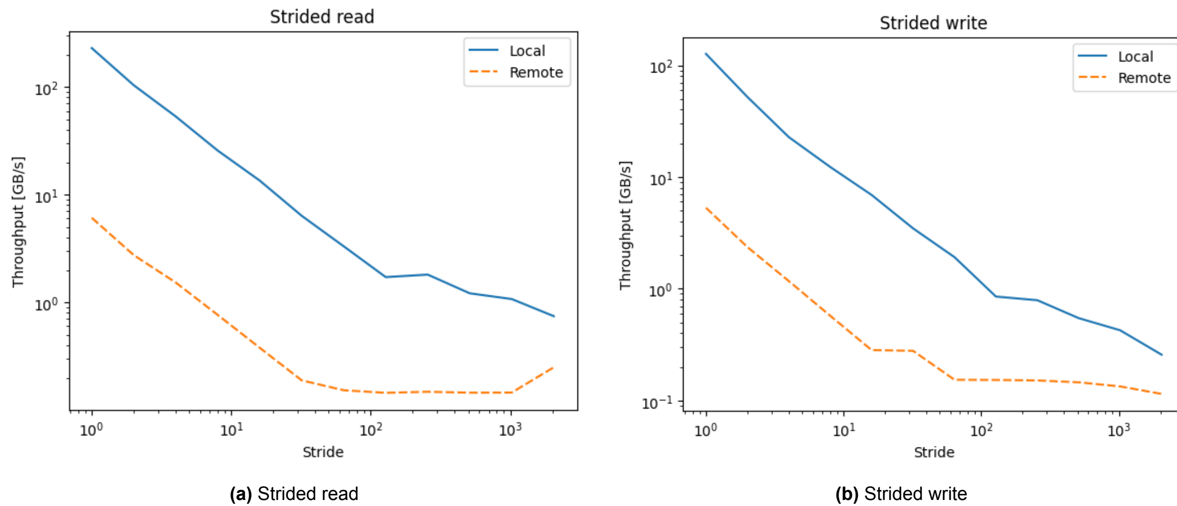
What we can clearly see here is that the ThymesisFlow link is quickly saturated when using data types wider than 8 bits. We also can see that the maximum bandwidth for ThymesisFlow reads and write are approximately the same. As it is expected reads to be much quicker, we can probably say the bottleneck is not the memory on the borrower side, but rather the max bandwidth of the ThymesisFlow 100Gbit link.

In the original paper the authors used 4 threads to run the STREAM benchmarks [1]. From the figure we can see why. Using 4 threads is already able to saturate the memory bandwidth to the near maximum bandwidth possible.

5.6.2. Strided access patterns

The easiest access pattern to read from memory is the serial case. Reads are done in memory from the beginning to end of memory one by one. The reason this is the fastest is because of a processor feature called the *prefetcher*. Let's say we read a memory address, the processor checks if it is in the CPU cache, if it is not it will fetch the data from memory. Fetching from cache is fast, if it is not in cache, we need to do a slow memory access. The CPU prefetcher logic tries to guess which data is going to be used by the application, and fetch and stores the cache lines it thinks are going to be needed beforehand. The logic for this in the Power9 processor is very complex, and it is not publicly known how it exactly works, however it is possible to measure its impact using this test.

These strided access patterns are common for data analytics pipelines. For example if we were to add up all values in a column, this would be a serial access pattern of stride one in which every value is visited once, and after each other. Bigger strides are common for joined data, or table like data where every x bytes contains a value of a row.



(a) Strided read

(b) Strided write

Figure 5.3: Read and write performance for differing stride access patterns

To get a feel how the Power9 processor behaves under different access conditions we devise a test setup to test how reliable the prefetcher is. Interesting here is that because ThymesisFlow makes use of OpenCAPI, the prefetcher logic is also used to fetch remote memory. This is a big advantage over some other rDMA technologies where data is only fetched when requested [23]. Specifically benchmarks which access data in a strided manner were created. Strided meaning that every X element is accessed. In the serial case, this means a stride of 1. We read for example the data at addresses: 0x00, 0x01, 0x02, ... For a stride of two we have an access pattern looking like: 0x00, 0x02, 0x04, ...

Because ThymesisFlow operates on 128-byte cache line size, data is over-fetched when reading for example the beginning of a cache line. When an application reads a single byte from memory for example, the full 128-bytes which contain that byte are fetched and stored in cache. Any accesses afterwards that fetch a byte within the previously fetched cache line will simply be returned from the CPU cache. Strided access patterns are a more difficult case, when using a stride of more than 128-bytes, every access will need to retrieve a new cache line through the ThymesisFlow link. And thus more heavily stressing the bus. The prefetcher plays an important role in this case, as there is a lot of performance to be gained if the prefetcher correctly predicts which cachelines are going to be fetched.

This results in the throughput measurements shown in Figure 5.3. In both figures we can see the prefetcher logic at work. For both remote and local memory accesses, we have that the throughput decreases for a higher stride, but does not completely diminish at the 128-byte mark.

If there was no prefetcher logic we would see a more ”flat” graph. The pattern in which data is fetched will not influence the throughput, as every data access will not be prefetched in cache, and will need to be accessed in the slower memory devices.

5.7. Concluding results

As stated before the initialization steps are the only overhead present caused by the coherency protocols described in this thesis. In Table 5.5 we can see all these measurements added up, these represent the time needed to initialize a full table. After an object is initialized, and made to be known on all other nodes, the only bottleneck then is the ThymesisFlow link. Several access patterns were analyzed to determine how the system will behave.

As a general statement on ThymesisFlow, we can say the maximum bandwidth is around 5.3 GB/s for both reading and writing. This is relatively quickly achievable with only 4 threads writing continuously. A more strided access pattern will still make use of the cache prefetcher logic, but will incur the same type of performance hits as a local memory access.

Part	Time [ms]
Malloc request	4,99
Remote pre-write flush (gRPC call + flush)	51.84
Write to remote	180
Flush local write cache	60.32
Serialize table descriptor	0.058
Send table descriptor to other nodes	3.23
	300.44

(a) Create table in remote memory

Part	Time [ms]
Malloc request	3.13
Flush borrower caches	51.84
Write to local	10
~~Flush local write cache~~	
Serialize table descriptor	0.058
Send table descriptor to other nodes	3.23
	68.26

(b) Initialization table in local memory. Local write flush not necessary, as any remote reads from this memory are snooped by OpenCAPI link. See subsection 2.8.1.

Table 5.5: Initialization overhead when creating a table in local vs remote memory. Table used is a 1GiB table with uint64 data elements in it

6

Recommendations and Future work

In this chapter a retrospect is given on the design decisions made in this thesis. Some designed systems could be extended or changed to be easier. But also some limitations of ThymesisFlow and Arrow can be resolved without work arounds if the architectures would be changed.

6.1. Instead of flushing, use write-through pages for writing to remote memory

Flushing protocols have been devised which turn the partial coherency guarantees of ThymesisFlow, into full coherency guarantees. This is done by flushing all touched memory regions, to ensure no writes are left hanging in the CPU cache. This is a problem when a node writes into a remote memory region, we need to ensure that the written data is actually written to the underlying remote memory.

An alternative approach could have been to map the memory region using *Write-Through memory pages*. When a page is marked using this mode, store instructions on this page cause the data to be written to both memory as well as the cache. A possible solution would be to have all pages which are remote, to be mapped as write-through. As every time a node writes to remote memory, data will *always* have to be written through to remote. The performance increase for this will be that the underlying memory will be written to memory, *during* the execution of the write instructions. Instead of having an idle period for the caches to fill up, then writing uncachable store operations to memory, and then having to flush all data blocks.

Future developments on the Arrow ThymesisFlow combinations are recommended to look into this optimization. A possible caveat is the ThymesisFlow stack not supporting this feature. There is a very clear warning in the ThymesisFlow documentation stating changes to the mapping functions will cause program and hardware crashes.

6.2. Method of Arrow allocations

Arrow has only one way in which objects are placed in memory. Data buffers are always stored in local, kernel and libc decided, memory regions. When an application wants to then share the Arrow object, it *has* to copy the object into a shared memory region.

For example, to allow for a local allocated object to be shared with a program running in a different process, a so called *RandomAccessFile* is created, a shared memory region which is mapped to both processes virtual address space. Arrow then has methods to write an Arrow object into this region, causing a copy without serialization to happen to this region. A receiving node is then able to create a local instance of the Arrow object. This Arrow object is able to be read from the shared memory region, without having to serialize or copy data. This copy operation is unnecessary, Arrow could allow for instantiating objects in this RandomAccessFile directly, omitting the copy operation.

Ideally Arrow would extend its MemoryPool functionality to allow for direct allocations in the RandomAccessFile. A possible solution would be to not only be able to pass MemoryPools to an Arrow object creation function, but also a RandomAccessfile, in which memory allocations are done inside of.

If this feature were to exist, it could easily be instrumented to work in remotely mapped ThymesisFlow memory.

6.3. More supported data types in table descriptor serializer

In this thesis code was used by the Fletcher project to serialize RecordBatch table descriptors. The code for this is quite limited in the supported data types, it only supports fixed width columnar formatted data. The power of Arrow though is that it is able to store complex data structures in a unified in-memory data format.

Making the serializer functions support all data types of Arrow would allow for a broader adoption, and allow for more complex disaggregation schemes.

6.4. Integrate with Arrow Acero execution graphs

Apache Arrow has an execution graph engine which allows for execution steps to be defined on big data, to be executed in one go. The system allows for inputting Arrow tables as a source for the execution graph. This gives rise to be able to source the in this thesis described HugeTable objects, into the execution engine.

Memory disaggregation could be a very interesting use case to combine with the Execution plan engine. As memory is freely shareable, Acero could for example execute different steps of a plan across a cluster, sharing Arrow data using the ThymesisFlow Arrow combination. Acero would need to be extended to not only execute an execution graph, but should also be able to serialize the graph and send parts of it to other nodes. The engine would coordinate the memory storage locations in ThymesisFlow memory, and take input data through ThymesisFlow, and output data through ThymesisFlow memory.

6.5. Distributed Query languages

Systems such as Apache Spark allow for defining a single program, which will be executed on a cluster of servers. One of the major bottlenecks present in Apache Spark data pipelines is that certain join operations are very network intensive. One such operation is a "wide shuffle". A wide shuffle is one where the output variables are heavily dependent on a big part of the input data. Compared to for example a narrow shuffle where an output value has only a handful of input values. An example of a wide shuffle is a "group by" operation, an example of a narrow shuffle is for example a filter or map operation.

Spark has some complex systems to execute these dependencies, but usually it is up to the developer to prevent as many wide shuffle dependencies as possible. With ThymesisFlow it is potentially possible for wide shuffle dependencies to be executed faster. Instead of having to broadcast all data to every node, requiring expensive copy and serialization steps, data can be retrieved through the ThymesisFlow link without copying a full data blob, and Arrow can be used to omit the serialization step.

7 Conclusion

This thesis has shown that Apache Arrow can be used in conjunction with ThymesisFlow to allow for memory disaggregation in compute clusters. An overview was given on what ThymesisFlow is, how it uses OpenCAPI to allow for transparent memory accesses across compute nodes. Furthermore Apache Arrow was studied, and how it allows for data interoperability between different processes. The combination of these two technologies allows for a fully memory disaggregated system, where data no longer needs to be copied across compute clusters over ethernet links, memory will be better utilized, and applications can use remote memory as if it is local. ThymesisFlow is used to transparently communicate with remote memory, the power is that any memory instructions addressing a remote memory region, are transparently relayed to the corresponding remote memory bus. Apache Arrow is used as it has a standardized data format so that data created somewhere else, is still readable by any other processes wanting to operate on that data.

Getting these two technologies to work together however is a challenge. First of ThymesisFlow is designed for the situation where memory of a lender node is borrowed to a single borrower node. The lender does not read or write its borrowed memory as that would make the system cache incoherent. In the case of a fully disaggregated system we do want that, we want the borrowed memory to be writable and readable by both the lender, and multiple borrowers. Apache Arrow also needs be modified, Apache Arrow does give a standardized format to store in-memory data, but it does not provide code for serializing table descriptors. Given that a node has access to Arrow data, it also needs a table descriptor to understand that data.

Apache Arrow was modified to circumvent the cache coherency issues in ThymesisFlow, and changes were made to the Arrow C++ library to allow for easy operability by developers of applications. The modified Arrow library allows a user applications to not worry about cache coherency issues, it allows for state synchronization across disaggregation nodes, and it handles the communication of Arrow data between nodes. A noteworthy feature is that of the *Huge Tables*, described in section 3.6. As Arrow allows so called ChunkedArrays to consist of multiple Arrays, the modified library is able to place multiple Arrays across a compute cluster. These separate Arrays, can then be combined into an Arrow ChunkedArray which spans multiple memory banks within the cluster. This means an array is no longer limited to the size of a single node, but can now be as big as the sum of all memory banks in a disaggregated cluster.

Furthermore, an analysis was given on the performance of a) the developed systems, and b) the ThymesisFlow memory bandwidth characteristics. Due to Apache Arrow objects being immutable, the cache coherency protocols only need to be employed during the initialization of Arrow data. After initialization the only performance penalty is that incurred by the ThymesisFlow system. For a developer to choose the combination of Arrow and ThymesisFlow, not only the overhead of initialization is important, but also the memory bandwidth for remote memory. An analysis was therefore also given on how several access characteristics influence memory bandwidth in the ThymesisFlow system. Namely: Read/Write, Bit widths, Thread counts, and finally strided access patterns. It was found that read and

write performance of ThymesisFlow memory is saturated at around 5.1GB/s, and decreases when using heavily strided access patterns. Furthermore, 4 threads was already enough to saturate the read and write memory queues of the internal OpenCAPI/ThymesisFlow bus. An analysis using Linux perf tools was used to indeed confirm the benchmarks were memory limited, not compute limited.

The research question formulated at the beginning of this thesis can be answered in a positive manner.

Research Question 1 Can memory disaggregation be made more accessible by combining ThymesisFlow and Apache Arrow? Using the abstraction layers the Apache Arrow libraries have built-in, software coherency systems are able to be implemented in such a way that the user application barely notices a difference with regular Arrow API calls.

Research Question 2 What cache coherency issues does ThymesisFlow have when multiple nodes access the same memory region? Can we work around those issues? ThymesisFlow was never designed for a situation where multiple CPU caches need to be kept coherent. It was designed for the situation that only one CPU accesses remote memory. One of the main issues is that in the case of a write to memory, only the CPU cache of the writer is coherent, any CPU caches which are not that of the writer, become incoherent. This can be worked around by first emptying all CPU caches by using flush operations, before memory is written to. After this the writer needs to ensure data is not left in its own cache by flushing memory to the remote memory.

Research Question 3 Is Apache Arrow a good fit as a memory disaggregation friendly data format? A standardized data format is a logical first step in creating a memory disaggregated system. After all, all nodes in a cluster should be able to understand data for data to be truly disaggregated. The promise of Arrow being fully zero-copy however, does not hold in all situations. Arrow does not allow already initialized objects to be shared with other processes, they have to be copied after initialization to a shared memory space. Objects can also not be instantiated in shared memory spaces, they have to be instantiated locally first. This thesis has modified the Arrow code to *support* instantiating objects in remote memory.

References

[1] Christian Pinto et al. "Thymesisflow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation". In: vol. 2020-October. Explanation of the ThymesisFlow framework. OpenCAPI with a software stack allowing for pooling of remote system memory. IEEE Computer Society, Oct. 2020, pp. 868–880. ISBN: 9781728173832. DOI: 10.1109/MICRO50266.2020.00075.

[2] Mohammad Ewais and Paul Chow. "Disaggregated Memory in the Datacenter: A Survey". In: *IEEE Access* 11 (2023), pp. 20688–20712. ISSN: 21693536. DOI: 10.1109/ACCESS.2023.3250407.

[3] Muhammad Tirmazi et al. "Borg: The next Generation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387517.

[4] Huaicheng Li et al. "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms". In: 2 (23). DOI: 10.1145/3575693.3578835. URL: https://doi.org/10.1145/3575693.3578835.

[5] *TOP500 | June 2023*. URL: https://www.top500.org/lists/top500/2023/06/.

[6] *The Beating Heart of the World's First Exascale Supercomputer - IEEE Spectrum*. URL: https://spectrum.ieee.org/frontier-exascale-supercomputer.

[7] *HPE Slingshot Interconnect – Your HPC Networking Solution | HPE*. URL: https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html.

[8] *TOP500 Highlights | June 2016*. URL: https://www.top500.org/lists/top500/2016/06/highlights/.

[9] Clemens Lutz et al. "Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects". In: (). HPI was involved in this paper.Describes NVLink architecture + cache coherence structure. DOI: 10.1145/3318464.3389705. URL: https://doi.org/10.1145/3318464.3389705.

[10] Robin Abrahamse, Akos Hadnagy, and Zaid Al-Ars. "Memory-Disaggregated In-Memory Object Store Framework for Big Data Applications". In: Main paper. IEEE, May 2022, pp. 01–07. ISBN: 978-1-6654-9747-3. DOI: 10.1109/IPDPSW55747.2022.00211. URL: https://ieeexplore.ieee.org/document/9835332/.

[11] Kevin Lim et al. *System-level Implications of Disaggregated Memory*.

[12] OpenPOWER. *POWER9 Processor User's Manual*. v21. IBM, 2019.

[13] *IBM AIX instruction manual*. URL: https://www.ibm.com/docs/en/aix/7.1?topic=memory-program-address-space-overview.

[14] *Man page for Linux mmap*. URL: https://man7.org/linux/man-pages/man2/mmap.2.html.

[15] Amir Roozbeh et al. "Software-defined 'hardware' infrastructures: A survey on enabling technologies and open research directions". In: *IEEE Communications Surveys and Tutorials* 20 (3 July 2018), pp. 2454–2485. ISSN: 1553877X. DOI: 10.1109/COMST.2018.2834731.

[16] Archit Patke et al. "Evaluating Hardware Memory Disaggregation under Delay and Contention". In: IEEE, May 2022, pp. 1221–1227. ISBN: 978-1-6654-9747-3. DOI: 10.1109/IPDPSW55747.2022.00210.

[17] *gRPC: A high performance, open source universal RPC framework*. URL: https://grpc.io/.

[18] *Protofbuf: language-neutral, platform-neutral extensible mechanisms for serializing structured data*. URL: https://protobuf.dev/.

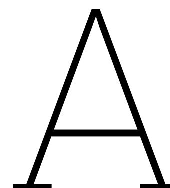
[19] *Fletcher: A framework to integrate FPGA accelerators with Apache Arrow*. URL: https://github.com/abs-tudelft/fletcher.

[20] OpenPower foundation. *Power ISA™ Version 3.1*. 2020.

[21] Embedded Artistry LLC. *Malloc: First-fit Free List*. https://github.com/embeddedartistry/embedded-resources. 2021.

[22] *OpenPOWER Summit NA 2019: Thymesis-P: An Approach to Rack-scale Disaggregation Over OpenCAPI*. URL: https://www.youtube.com/watch?v=XcjRL3Lh8Ig.

[23] Qirui Yang et al. "Performance Evaluation on CXL-enabled Hybrid Memory Pool". In: IEEE, Oct. 2022, pp. 1–5. ISBN: 978-1-6654-5408-7. DOI: 10.1109/NAS55553.2022.9925356.



CPython list data structure

CPython data structure for storing lists in Python. Pointer array where every pointer points to an allocated PyObject list element. Because every PyObject can have its own type, every element in the list can also have its own type.

```
typedef struct {
    PyObject_VAR_HEAD
    /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
    PyObject **ob_item;

    /* ob_item contains space for 'allocated' elements.  The number
     * currently in use is ob_size.
     * Invariants:
     *     0 <= ob_size <= allocated
     *     len(list) == ob_size
     *     ob_item == NULL implies ob_size == allocated == 0
     * list.sort() temporarily sets allocated to -1 to detect mutations.
     *
     * Items must normally not be NULL, except during construction when
     * the list is not yet visible outside the function that builds it.
     */
    Py_ssize_t allocated;
} PyListObject;
```

B

Bring-up of HPI ThymesisFlow setup of Hasso Plattner Institute

The ThymesisFlow setup by the Hasso Plattner Institute consists of two Power9 ic922 servers. These servers have the required hardware and firmware for a 2 node ThymesisFlow server. Node 04 is a memory lender node, node 03 is the borrower compute node. Researchers at HPI have developed a handy kernel module set to make the mapping of remote memory even easier. After initialization, there exists a `/dev/mishmem-s1` file on both nodes, which can be easily mapped to a userspace program with the well known mmap syscalls. The initalization steps are described below.

Load kernel module on node 03: `sudo insmod /opt/mishmem/mishmem-s1/mishmem-s1.ko`

Check if it has been loaded correctly by `ls -l /dev/mishmem-s1`. There should be a file mishmem-s1

On node 04 check if there is a memory map file: `ls -l /dev/mishmem-s1`. If the file is not present, or if it is too small, you can create/extend it with: `sudo /opt/thymesisflow/init_shmem_file.sh 32768`. Where the number is the amount of 1MB blocks you want the file to contain. CAREFUL: This is the number of 1MB blocks, if you want 1GB, allocate 1024 blocks!

On node 04, the memory lender, initialize ThymesisFlow with the following command. Size is in this case bytes.

```
/opt/thymesisflow/bin/thymesisf-cli \
        attach-memory \
        --afu IBM,RMEM \
        --cid 1 \
        --size 34359738368 \
        --port 2
```

On node 03, the compute node, the borrower, execute the following. `--no-hotplug` is required as it is ”less” stable if it is enabled.

```
/opt/thymesisflow/bin/thymesisf-cli \
        attach-compute \
        --afu IBM,RMEM \
        --cid 1 \
        --size 34359738368 \
        --port 2 \
        --ea 0x100000000 \
        --no-hotplug
```

The link has now been established! We can execute a test by writing to the memory and reading from it on the other side. On the 04 node, write some data from the memory side:
`dd of=/dev/mishmem-s1 bs=1 conv=notrunc`

Type something, then hit Ctrl-D to finalise. Note that at the time of writing, block writing is not supported, writes should happen byte per byte using `bs=1`. Also pass the `conv=notrunc` to prevent dd form truncating the memory file to the input you gave it.

On the compute node read this memory by executing the below sample C program:

```
1  #include <stdio.h>
2  #include <sys/mman.h>
3  #include <fcntl.h>
4
5  int main (void) {
6          int fd = open("/dev/mishmem-s1", O_RDONLY);
7          if (fd == -1) {
8                  perror("open");
9                  return -1;
10         }
11
12         void *map = mmap(NULL, 0x1000, PROT_READ, MAP_PRIVATE, fd, 0);
13         if (map == MAP_FAILED) {
14                 perror("mmap");
15                 return -1;
16         }
17
18         printf("%s\n", (char *) map);
19
20         return 0;
21 }
```

If anything is wrong, the ThymesisFlow kernel driver may be in an incorrect state, or the FPGA may have crashed. In my code this usually resulted in `Bus error (core dumped)`. Whatever may be the case, sometimes the setup needs to be restarted. The nodes can be each restarted with `sudo reboot`.