Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Magne Angvik Hovdar

Video distribution using PCI-Express

Master's thesis in MTTK
Supervisor: Sverre Hendseth
Co-supervisor: Hugo Kohmann
June 2023

NTNU
Norwegian University of
Science and Technology

Magne Angvik Hovdar

Video distribution using PCI-Express

Master's thesis in MTTK
Supervisor: Sverre Hendseth
Co-supervisor: Hugo Kohmann
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Video distribution using PCI-Express

Author:
Magne Angvik Hovdar

Supervisor:
Sverre Hendseth

Master thesis
Department of Engineering Cybernetics
Norwegian University of Science and Technology

June 4, 2023

Preface

This technical paper details the design and implementation of a video sharing system utilizing PCI-Express, a high-speed serial expansion bus standard widely used in computers for adding peripheral devices. The paper is a collaboration with Dolphin Interconnect Solutions and is a showcase of the potential of PCI-Express interconnect technology. The application uses Dolphin's software and hardware to accomplish this.

The intended audience for this paper includes both practitioners and researchers in branches related to high-performance computing. A basic understanding of computer hardware and data transmission protocols is assumed, as well as knowledge regarding basic low-level programming techniques.

Hopefully, this paper offers valuable insights into the implementation as well as inspiration to further build on this technology.

Acknowledgements

I would like to thank the people at Dolphin Interconnect Solutions for providing an interesting task as well as for their general helpfulness. In particular Hugo Kohmann who has co-supervised the project.

I would like to thank my supervisor Sverre Hendseth for providing helpful and original hints on how to create and structure a technical paper.

Executive summary

This technical paper presents the development and implementation of a pioneering video transfer application leveraging the high-speed serial expansion bus standard, PCI-Express. The project was undertaken in collaboration with Dolphin Interconnect Solutions, who's products enable the use of PCI-Express technology for higher-level applications such as video transfer. In particular, this application focuses on "multicast", a reflective memory implementation in hardware.

The problem at hand was to create a robust and efficient video sharing application that showcases the features provided both by PCI-Express and the Dolphin products. The development process involved researching, planning, testing, and implementation, all of which are thoroughly documented in the subsequent sections of this paper.

The result is a fully functioning application that meets the initial objectives and provides insights for how the technology can be developed even further. It provides a solution that is first of it's kind.

One key point in this project is that is it designed to be built further upon in the future by making code and implementation as accessible as possible. By adopting this approach, we have ensured that our application can be further expanded and refined by others in the future, both for academic and professional reasons.

This paper should serve as a comprehensive guide for those interested in the technicalities of PCI-Express, Dolphin products and particularly how the multicast technology can be leveraged for advanced applications. Hopefully, this lays the groundwork for future advancements within high-performance interconnected video systems.

Sammendrag

Denne tekniske rapporten presenterer utviklingen og implementeringen av en videooverføringsapplikasjon som utnytter høyhastighetsbussen PCI-Express på en måte som aldri før har blitt gjort. Prosjektet ble utført i samarbeid med Dolphin Interconnect Solutions, og deres produkter som muliggjør bruk av PCI-Express-teknologi for mer avanserte applikasjoner slik som videooverføring. Denne applikasjonen fokuserer på ”multicast”, en reflekterende minneimplementasjon i hardware.

Problemstillingen var å skape en robust og effektiv videodelingsapplikasjon som fremhever funksjonaliteten som finnes både i PCI-Express og Dolphin-produktene. Utviklingsprosessen involverte forskning, planlegging, testing og implementering som er grundig dokumentert i de påfølgende delene av denne rapporten.

Resultatet er en fullt fungerende applikasjon som oppfyller målene fra problemstillingen og samtidig gir innsikt i hvordan teknologien kan utvikles videre. Løsningen som her fremstilles er den første av sitt slag.

Et nøkkelpunkt i dette prosjektet er at det er designet for å bygges videre på i fremtiden ved å gjøre kode og implementasjonsmetode så tilgjengelig som mulig. Ved å gjøre det på denne måten, er målet at applikasjon kan utvides og forbedres av andre i fremtiden, både for akademiske og profesjonelle formål.

Denne rapporten bør fungere som en dekkende guide for de som er interessert i teknikken bak PCI-Express, Dolphin-produkter og spesielt hvordan multicast-teknologien kan utnyttes til å lage avanserte applikasjoner. Forhåpentligvis legger denne rapporten grunnlaget for fremtidige fremskritt innen sammenkoblede videosystemer med høy ytelse.

Table of Contents

List of Tables

List of Figures

Abbreviations

Abbreviation	Description
PCI	Peripheral Component Interconnect
MPS	Maximum Package Size
GPU	Graphical Processing Unit
CRC	Cyclic Redundancy Check
TLP	Transport Layer Packet
SIA	Self-Installing Archive
AGP	Accelerated Graphics Ports

1 Introduction

This masters thesis project presents the work on a video distribution application that uses the bus technology known as PCI-Express (Peripheral Component Interconnect Express) to distribute video to several devices at once. The application presents a new bridging of existing technologies that provides opportunities for innovation and increase in performance. As demands for throughput and reliability increase in general computing, technology such as the one developed by Dolphin Interconnect Solutions for general purpose high performance data transfers over PCI-Express extend their areas of appliance. In collaboration with Dolphin and their PCI-Express based stack, this paper will tackle video transfers using PCI-Express and reflective memory technology.

1.1 Context

This project is a collaboration between the author and Dolphin Interconnect Solutions (hereby: ”Dolphin”). Dolphin is a Oslo-based company that are and have been designing and manufacturing PCI-Express hardware and software products for more than 25 years (Dolphin (2015)). Their products are used in high-performance systems within many sectors such as defence, medical and automotive technology. Dolphin currently has a demo program for transferring video from one device to another using their products that could be extended extensively in order to demonstrate more of the capabilities their products provide.

PCI-Express has been used for application specific purposes for a long time (IEEE (2022)), most commonly known for connecting the GPU (Graphical Processing Unit) to your computer. PCI-Express is however, a fast, reliable and continuously evolving technology that can be used for more generic purposes as well (Dolphin (2015)). Dolphin’s unique software and hardware products allows for utilization of the speed and robustness the PCI-Express standard provides even for general purpose tasks. This allows this project to tackle problems not previously solved and provides demonstration of Dolphin product capabilities at the same time.

1.2 Motivation

Dolphin Interconnect Solutions have provided this problem as a means to increase the utility of their products. As the use cases for PCIe technology expand and increase, so does the potential of Dolphins technology. As an extension to their existing technology, Dolphin would like to demonstrate how PCI-Express can be used for sophisticated video transferring and processing techniques.

For the aforementioned sectors who utilize PCI-Express compliant devices in their systems, an extension of the currently available features means reducing the amount of different technologies that need upkeep. In many cases, there should also be a potential increase in speed and reliability by opting to use this method for video applications as well. In order to figure out the specific improvements that can be provided by using Dolphin products for video specific applications, research is required.

1.3 Problem description

The problem presented by Dolphin Interconnect Solutions is found in Appendix A and consists of building software using Dolphin's products that allows video transfer in real-time from one device to one or more connected clients. The program should be able to demonstrate advanced PCI-Express techniques in a way that exceeds the capabilities of their current solutions.

The problem description is presented as a choice between different potential expansions where not all are assumed to be completeable within the timespan of a masters thesis. It is the task of this exercise to identify which expansions are feasible to integrate within the time period of the thesis. The decisions and acquired knowledge regarding what to integrate, if it would work and how it could be integrated into a larger project will be part of the final product. The proposed features in Appendix A are summed up here:

One possible expansion is adding support for ”multicast”, a Dolphin broadcast option that allows simultaneously writing to memory in multiple computers at once. This can be done with or without a GPU as part of the supply chain to process the video before it is shown on the client devices.

A secondary option is to add support for direct transfer from a camera into the device(s). The camera feed could be send directly to displaying nodes over multicast or to a GPU for processing.

Due to circumstances explained in section 1.4, this technical paper focuses on the first option of creating an application that distributes video to several nodes using multicast. The application should be implemented with the other extensions in mind and any advances towards GPU or camera integration will be a bonus for the result.

1.4 Limitations

The existing demo application deployed by Dolphin is not open-source and in discussion with Dolphin, an agreement was made that the version developed in this project should be open-source and made available for further academic research.

Consequently, this project is not an expansion to existing demonstration capabilities but rather a complete rewrite of a demo application from the bottom up. This choice comes with a significant increase in development time but is ultimately regarded as a worthwhile option because it makes the technology more available for future academical and professional interest.

The PCI-Express camera described in the problem description is not manufactured by Dolphin and was not available for testing purposes during the timespan of this project. In light of this, the development was shifted more towards the other options presented in the problem description. The development process nevertheless considered the camera integration as an option for the future.

1.5 Structure of the report

The report is structured as follows:
An introductory part is followed by a background chapter that sums up relevant literature and lays the theoretical foundation for the technology and tools that are needed to create the application. This includes an introduction to PCI-Express tools that are needed and used as well as information regarding processing and displaying.

The goal of chapter 2 is to contain all fundamental knowledge acquired prior to developing the application. It should sufficiently cover the knowledge needed to comprehend the parts following it.

All chapters following the background chapter are based on original work and findings. This part starts with a methodical approach spanning the chapters ”Specification”, ”Design”, ”Implementation” and ”Testing”. After this follows results, reflections and future work which are especially important considering the open nature of the problem description.

2

Background

This chapter contains an introduction to PCI-Express tools that are needed and used for the resulting application. This includes but is not limited to how segments are created, how memory physically present on another computer can be mapped into a process' memory and how this memory can be read and written to. Moreover, an introduction to used frameworks and common techniques such as multithreading and using mutexes is presented as well as introductions to selected languages, frameworks and methods.

The research presented in this article is done using software and hardware provided by Dolphin Interconnect Solutions. The provided tools from Dolphin give the software developer a programming interface for performing PCI-Express operations such as remote memory access and high-speed transfers of data. The interface provided by Dolphin for this purpose is the SISCI API which is used in this application.

PCI-Express (Peripheral Component Interconnect Express) is a high-speed computer bus standard that is used to connect hardware devices, such as graphics cards, network cards, and storage devices, to a motherboard. It was introduced in 2004 as a replacement for the older PCI and AGP bus standards, and has since become the most widely used expansion slot interface in modern desktop and server computers. PCI-Express technology offers high bandwidth, low latency, and improved power management. This has made it an essential technology for high-performance computing applications, including gaming, scientific research, and artificial intelligence. As the demand for higher performance computing has increased, so has the use-cases for PCI-Express technology.

2.1 Related works

As far as the author aware, there have been no comparable attempts at doing what this project attempts to accomplish. Additionally, the problem description provied in Appendix A clearly sets the scope of the task outside of PCI-Express technicalities. With the exception of some particular PCI-Express knowledge in order to determine throughput performance, the PCI-Express specifics that Dolphin products used are built on will be regarded as outside the scope of the project. The relevant literature is therefore limited mostly to the documentation of the frameworks utilized which in the case of SISCI is quite extensive.

Some of the examined literature include two theses that utilize Dolphin technology for slightly different goals. (Cubedo (2021)) used Dolphin PCI-Express technology to improve the performance of machine learning models. The author developed a proof of concept plugin for the NVIDIA Collective Communication Library (NCCL) that enables inter-machine PCI-Express communication using Dolphin NTB adapters. Although the goals and methods of the thesis were substantially different, there are still some key takeaways in how the performance gain can be measured for instance.

(Skrede (2022)) worked on message passing interfaces over PCI-Express using Dolphin technology to universalize high performance computing. Again, the resulting performance could be helpful for benchmark comparisons but the tasks are fundamentally very different. The methodology used is also found in (Dolphin (2017b)) which will be the preferred source.

There is also a thesis by (Lye (2010)) that implements camera support over PCI-Express for a system not relying on Dolphin technology. Because the camera support was discarded as an option for this application, this did not become relevant.

This task is unique in the way that it is heavily based on ”multicast” which means the whitepaper (Dolphin (2017a)) is essential. It provides comprehensive information on transmitting and reading data from reflective memory, highlighting the significant benefits of using PCI-Express. The document also includes detailed hardware configuration and installation instructions, a reflective memory comparison, and future plans. Additionally, it provides code examples for the SISCI API. The whitepaper serves as a valuable resource for understanding the application and benefits of PCI-Express reflective memory in data transmission.

The research for this task has greatly involved Dolpin Interconnect Solutions and the people working there in order to categorize which ideas and expansions are viable and not. In general, research has been specific to development and revolved around the tools necessary as well as insight into the capabilities supplied by Dolphin’s products.

2.2 PCI-Express standard and transfer speed

Having a basic knowledge of the PCI-Express generations and the transfer speeds associated with them is somewhat integral to the ability to measure performance in a PCI-Express system.

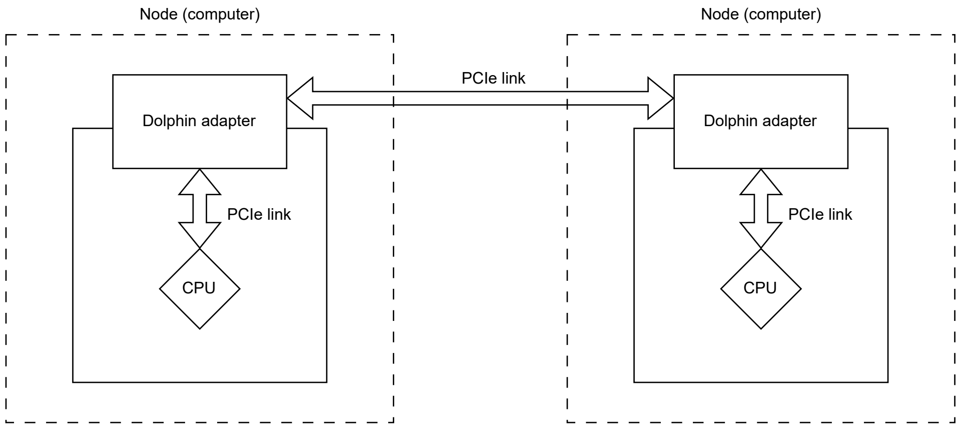


Figure 2.1: A PCI-Express link using Dolphin products also relies on the PCI-Express link between the adapter card and the CPU.

Generation	Data Rate	Lane Bandwidth (1 dir.)	Maximum Bandwidth	Release Date
1.0	2.5 GT/s	250 MB/s	4 GB/s	2003
2.0	5 GT/s	500 MB/s	8 GB/s	2007
3.0	8 GT/s	985 MB/s	15.75 GB/s	2010
4.0	16 GT/s	1.97 GB/s	31.5 GB/s	2017
5.0	32 GT/s	3.94 GB/s	63 GB/s	2019
6.0	64 GT/s	7.56 GB/s	121 GB/s	2021
7.0	128 GT/s	15.13 GB/s	242 GB/s	TBD

Table 2.1: This table presents key characteristics of different generations of PCI-Express interfaces. ”Data rate [GT/s]” refers to the theoretical maximum data transfer rate per lane, measured in GigaTransfers per second. A transfer is essentially a change in the binary state, from 0 to 1 or vice versa. ”Lane bandwidth” refers to the actual data transfer rate per lane, after accounting for encoding overhead. ”Maximum bandwidth” refers to the total data transfer rate for a full-sized x16 slot in one direction.

As specified in (PCI-SIG (2006)), the PCI-Express 2.0 standard provides a maximum theoretical bandwidth of 5.0 GT/s (GigaTransfers per second) per lane. After accounting for the 8b/10b encoding scheme used in PCI-Express 2.0, this translates to approximately 500 MB/s (Megabytes per second) of actual data transfer rate per lane. A standard full-sized PCI-Express slot (x16) can therefore provide a maximum theoretical bandwidth of 8 GB/s (Gigabytes per second) in each direction (upstream and downstream), for a total of 16 GB/s.

Equivalently, as specified in (PCI-SIG (2010)), the PCI-Express 3.0 standard provides a maximum theoretical bandwidth of 8.0 GT/s per lane, which translates to approximately 985 MB/s of actual data transfer rate per lane, after accounting for the 128b/130b encoding scheme. A standard full-sized PCI-Express slot (x16) can therefore provide a maximum theoretical bandwidth of 15.75 GB/s in each direction, for a total of 31.5 GB/s.

The PXH830 supplied by Dolphin for the application uses PCI-Express 3.0 and supports the corresponding transfer speeds. However, as illustrated in Figure 4.1, the PCI-Express throughput of the system is dependent on both the link between the adapter cards in each node and the link on the motherboard between the adapter card and the processor. There is therefore no guarantee that the transfer speed will reach PCI-Express 3.0 standard speeds without having a bottleneck out of the processor. The PCI-Express physical links in are illustrated in Figure 4.1.

The PXH830 comes with four cables for interconnect which gives the full x16 speed when they are all connected. The performance can be scaled down to x8 or x4 by using only two or one cable(s), respectively.

2.2.1 Transaction layer packets and their performance effect

PCI-Express communication revolves around the concept of sending and receiving TLPs. These are the basic units of transfer on the PCI-Express bus, and they encapsulate the data being transferred along with some additional information necessary for delivering the data correctly.

As referenced in (PCI-SIG (2006)), TLPs are responsible for requests and completions in PCI-Express. A request TLP is sent by a requester (initiator), and a completion TLP is sent by a completer (responder). The maximum payload size (MPS) is a parameter that defines the maximum size of the data payload in a single TLP and can vary in size depending on the processor's capabilities. In general, regular processors have lower maximum payload sizes than FPGAs but it depends on the processor brand and model. The TLP includes several fields:

1. **Header**: This contains information about the type of transaction, the length of the data, and the source and destination of the transaction. The header is 20 bytes in size.
2. **Data**: This is the payload of the TLP, which can be up to 4096 bytes in size.
3. **Digest** (optional): This is a cyclic redundancy check (CRC) field used for error detection.

The MPS supported by the processor sets a limit on how much data can be transferred in a single TLP and therefore decides the data utilization of the TLP for the system.

2.3 The Ethernet standard as compared to PCI-Express

The table above is created using information found in (IEEE (2022)). It shows us the speed provided by the Ethernet protocol as a useful comparison to the performance we hope to achieve using PCI-Express alternatives. A general Ethernet link between home computers is usually of the standards 1000BASE-TX or 10GBASE-TX with their corresponding

Table 2.2: Ethernet Speed standards.

Version	Bandwidth	Standardized in	Cable
10BASE-T	10 Mbps	1995	UTP
100BASE-TX	100 Mbps	1998	UTP
1000BASE-T	1 Gbps	2003	UTP
10GBASE-T	10 Gbps	2008	Cat6a or better UTP
25GBASE-T	25 Gbps	2016	Cat7 or better UTP
40GBASE-T	40 Gbps	2017	Cat7 or better UTP
100GBASE-T	100 Gbps	2019	Cat8 or better UTP

speed statistics. We also have to consider that the actual bandwidths do vary based on factors such as the length of the cable, the network load and the type of traffic being sent. In general, differences in protocol overheads make the actual bandwidth very dependent on the data being sent which is something to consider when comparing different technologies.

2.4 Video transfer

The main comparison to a PCI-Express based video transfer application is the common method of using a TCP/IP-based stack. TCP/IP-based applications have a big advantage in their availability. Any device that supports internet connectivity can in theory partake in a video sharing application. The downside of this ”all-purpose” approach is speed and robustness. Depending on the physical link between devices, speed can be a varying factor. In addition to this there is a need for continuous TCP/IP calls during data transfers which are a limiting factor both for transfer speed and CPU usage.

On the other end there are embedded applications where specially designed protocols or protocol-less systems have the flexibility to allow speed and efficiency only limited by the hardware. This approach sacrifices generality and availability, and gives the opposite benefits and disadvantages to a high-level protocol like TCP/IP.

My approach uses the Dolphin hardware and software stack to bridge these options. By using Dolphin adapter cards and the software stack known as SISCI (Software Infrastructure Shared-Memory Cluster Interconnect) I can use **regular processor store and load commands** to transfer data from one computer to the other. This avoids the previously mentioned overhead of TCP/IP. When it comes to availability and generality, it is possible to use SISCI on a wide range of different hardware running different software and operating system configurations.

One SISCI feature that is especially useful for this purpose are multicast addresses which allows a single write operation to update memory on several other nodes through reflective memory hardware. This can be used to create a bandwidth-efficient sharing application for multiple clients.

2.5 Hardware

The drivers provided by Dolphin Interconnect Solutions drive an adapter board connected to the motherboard's PCI-Express slot. The board used is the PXH830 ((Dolphin, 2019)) which supplies a link layer connection between the nodes in the network. The board can be seen in Figure 2.2. The PXH830 allows for high-speed transfers with optical cables and provides DMA capabilities for efficient transfers of bigger sizes.

In the testing cluster used, the two PXH830 adapters (one in each computer) are connected with four cables providing a x16 link. They are configured to use their highest supported version of PCI-Express being version 3.0. As per the specification ((PCI-SIG, 2010)) this should allow for a theoretical maximum transfer speed of 16GB/s.

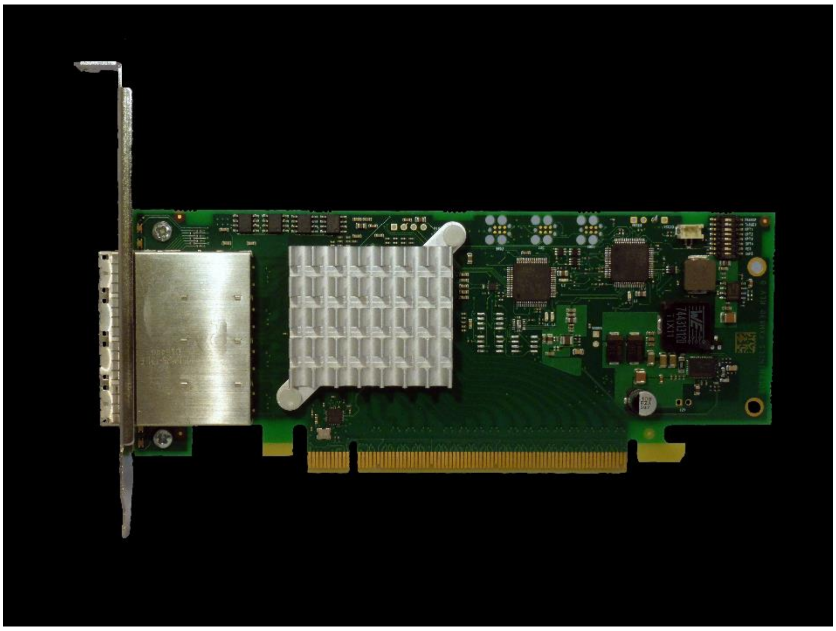


Figure 2.2: The Dolphin Interconnect Solutions PXH830 adapter card as pictured in its manual ((Dolphin, 2019)).

2.5.1 PCI-Express switch

Multicast is a SISCI operation where a memory segment is reflected in several computers. This allows a single processor write operation to alter memory on several nodes at once. This operation is enabled by a PCI-Express packet switch created specifically for PCI-Express clusters by Dolphin. The switch provided for this project is the MXS824 found at (Dolphin (2023)).

2.6 Dolphin software

The use of Dolphins PCI-Express software stack for video processing purposes is mostly uncharted territory. This task utilizes a combination of hardware and software provided by Dolphin to make advancements on this. SISCI, or (Software Infrastructure for Shared-memory Cluster Interconnects) is an API designed to utilize the functionality of the installed PCI-Express adapter board. It provides a library of SISCI functions that allows the programmer to utilize efficient PCI-Express functionality. The provided library communicates with the SISCI driver and provides ”made easy” ((Dolphin, 2017b)) PCI-Express capabilities.

2.6.1 Virtual devices

A SISCI virtual device is required for every distinct resource held by the running program. This means that a virtual device can only hold one of *each* resource with an example being one memory segment and one interrupt but not two of either. Functionally, this allows a virtual device to use all API functions available for a single connection. Multiple connections require one virtual device per connection.

The properties of a resource are collected in a descriptor and provided to the programmer through a *handle*. So in short, a virtual device is a communication channel with the driver that contains a maximum of one of each resource. The resources are accessible by the user though their provided handles.
In order to create a virtual device we need to initialize the SISCI library and then open a virtual device. This can be achieved using the following template:

```
SCIInitialize()
SCIOpen()
/*   Sisci functionality here    */
SCIClose()
SCITerminate()
```

With each virtual device needing to be created and removed using `SCIOpen` and `SCIClose`, respectively. Initialization and termination is necessary only once per program. Through virtual devices, remote memory can be mapped directly into the memory of the running process and accessed as if it were local, with SISCI drivers functioning as an intercepting layer for seamless integration.

2.6.2 SISCI segments

Remote memory access is handled using *segments*. A memory segment can be set available for access from another device creating what is a *local segment* from the perspective of the node that sets it available for others. Because memory allocation is a process that is handled differently in various operating systems, SISCI supplies specific functionality for allocating and preparing memory for segments. An operating system swapping process memory allocated in a traditional way could for instance be detrimental to a shared-memory application. The process of creating and preparing a segment for use on a physically separate device can be simplified to:

```
sci_desc_t v_dev;
sci_local_segment_t local_segment;
sci_error_t error;

SCIInitialize(...);
SCIOpen(&v_dev,...);
SCICreateSegment(v_dev, &local_segment, ..., &error);
SCIPrepareSegment(local_segment, ...);
SCISetSegmentAvailable(local_segment, ...);
```

where some required arguments have been omitted for the sake of simplicity but are specified in ((Dolphin, 2020)). The segment is created and memory is allocated in a safe way in accordance to the operating system with `SCICreateSegment`. `SCIPrepareSegment` maps the memory segment into a cluster-wide network address space allowing access to the segment from network adapters. This function also ensures that the memory is allocated safely and in compliance with the local operating system. `SCISetSegmentAvailable` allows importing nodes to connect to the segment and map it into their own memory. In combination with its counterpart, `SCISetSegmentUnavailable` it's possible to react based on remote processes accessing the segment.
A remote node trying to connect to a segment made available on another device can do so by representing the segment as a *remote segment* in its own memory. Data flow through segments are illustrated in Figure 2.3.

2.6.3 Client-server based memory operations

Transferring data in a cluster of computers connected by Dolphin adapters can be done in a pattern where the node that contains the physical data acts as a server and the nodes requesting the data act as clients. One standard way to transfer anything from a running server to a client requesting data is by using a doorbell mechanism in the following pattern:

1. The server makes a segment available for clients to notify their activity. This can for instance be configured as an interrupt.
2. The client makes a data segment available for connections.
3. The client connects to the server and notifies it that a file is requested transferred to the data segment made available.
4. The server connects to the client's available data segment and pushes the data (for instance using a DMA engine).

Once notified that a transaction is required, the server maps the available memory of the client into it's own memory space and transfers the data using regular processor I/O operations or provided DMA capabilities.

Processor In/Out operations (PIO)

The simplest way of writing to a memory address that is present on a different computer but mapped into the local system with a SISCI segment is by simply writing to it. While

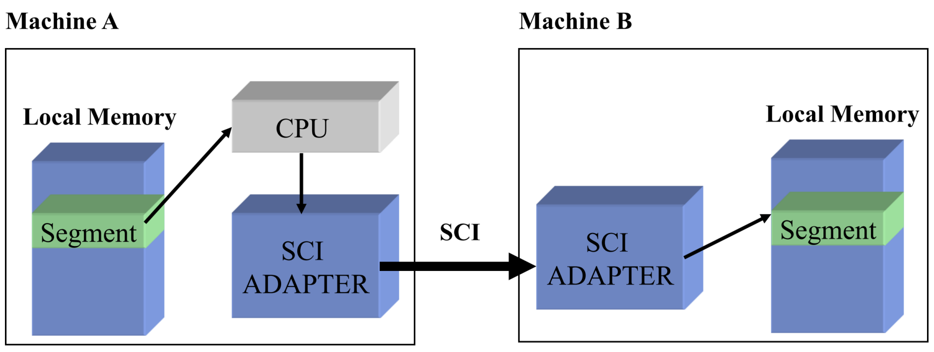


Figure 2.3: Illustrative picture of data flow from the running process through the Dolphin adapters provided by Dolphin Interconnect Solutions.

this can be done in any traditional way, the most efficient method should always be using `SCIMemCpy()` according to (Dolphin (2017b)). `SCIMemCpy()` uses PIO for data transfer but is optimized for the CPU and motherboard hostbridges. The `SCIInitialize()` call finds the optimal `SCIMemCpy()` and is required upon every SISCI program initialization.

Direct Memory Transfer (DMA)

The Dolphin adapter card PXH830 comes with a physical DMA that allows the use of SISCI DMA functions. The DMA can move data from a local segment to a remote one without taking up CPU resources, freeing the CPU to be used for other operations during the transfer. The logical separation between the two modes for optimal performance as supported by (Dolphin (2017b)) is to use the DMA for large data transfers and PIO for smaller amounts of data. This is because the DMA has a significant overhead, making it the faster option only at a certain transfer size. A DMA queue with the aforementioned capabilities is a SISCI resource and therefore needs a virtual device as described in subsection 2.6.1. However, as specified in that section, the same virtual device can be used to keep track of several different SISCI resources. Using the same virtual device that keeps the *handle* for the remote segment we want the DMA to transfer to, we can also initialize an accompanying DMA queue with it's own handle. The DMA queue handle `dma_queue_t` is initialized using `SCICreateDMAQueue()` and passing the adapter number for the adapter with the physical DMA engine. Once the queue is initialized, passing data using the DMA can be done in one of two ways:

- Using `SCIStartDmaTransfer()` which takes as it's input a local and a remote segment, transferring data from the local to the remote unless otherwise specified using flags.

- Using `SCIStartDmaTransferMem()` to transfer directly from a user-allocated buffer to a remote segment.

The DMA transfer has several synchronous and asynchronous options for validating data transfer as well as other options that will be omitted for the sake of conciseness but can be found in (Dolphin (2017b)). Instead, relevant options for testing purposes are presented when utilized in chapter 6.

2.6.4 Multicast

Multicast is a SISCI mechanism that allows for reflective memory, meaning a write operation can write to memory in several computers at once. This is done by assigning *multicast ID* groups to local segments and mapping remote segments on the senders end to these. The multicast mechanism handled by SISCI hardware meaning that writing to multiple addresses is as simple as a regular processor write operation.

2.7 Programming language and OS. specific features

Mutexes and semaphores are synchronization primitives used in multithreading programming to handle concurrent access to shared resources. A mutex is a binary flag that ensures mutual exclusion between concurrent threads on a single resource. When a thread acquires a mutex, no other thread can access the guarded resource until the original thread releases the mutex.

A semaphore is a more generalized synchronization tool that controls access to a set of resources instead of a single one. It maintains a count of available resources, and when a thread requests a resource, the semaphore's count decreases. If the count reaches zero, requesting threads block until a resource becomes available again.

When using C and Linux, these features are available through the POSIX threads (pthreads) library. The `pthread_mutex_lock` and `pthread_mutex_unlock` functions handle mutex operations, while `sem_wait` and `sem_post` handle semaphore operations. These mechanisms help prevent race conditions, ensure data consistency, and coordinate actions between threads.

2.8 Video processing

In order to create an application that can support the transfer of different types of video, as well as accommodate both file transfers and potential direct buffer writes from a camera source, processing is necessary. Some of the features needed for the functionality of the program are:

1. Breaking the video up into individual frames for data transfer
2. Separating video and audio data
3. Ability to break several types of video formats into one transferable datatype
4. Create displayable formats from raw image or video data

The video processing techniques needed for this project are relatively basic but should be able to extend into more advanced techniques if a GPU is utilized to process the data. Flexibility is therefore a main requirement for the processing method.

2.8.1 About the framework: FFMPEG

Because the project is open-source, the open-source video processing framework FFmpeg fits well with the task at hand. FFmpeg is a popular open-source multimedia framework that provides powerful tools for handling various multimedia formats. It supports a wide range of codecs, formats, and protocols, and can be used for encoding and decoding, of audio and video files. FFmpeg is capable of converting files between different formats, manipulating metadata, and extracting audio and video streams from file containers which will be very useful when processing a video stream over PCI-Express. It can be integrated into the application through its libraries and APIs although for the task at hand, the C libraries are preferred.

Relevant library methods and datastructures

We need to establish some terminology in order to make use of the FFmpeg library functions. Here follow the relevant video processing terminology:

- **Codec**: A program or device that encodes or decodes a digital data stream or signal. In the context of video files, a codec determines how the video is compressed. Common video codecs for video include H.264, HEVC, VP9, and AV1. FFmpeg supplies several codecs and is able to pick one that can decode our video based on it's format. This is essential for breaking down videos in different formats to the same, PCI-Express transferable format.

- **Decoding**: The process of translating a compressed video data stream back into a series of images (frames). The compressed video data is encoded in a specific format, and the decoder must understand this format to correctly translate it back into a watchable video.

- **Packets**: A unit of data that is transmitted over a network. In terms of video files, a packet typically contains a portion of the compressed video data. Packets are part of the ”container format” of the video file, which includes the video data, audio data, and other information like subtitles or metadata.

- **Streams**: A sequence of data that represents a particular element of the video file. For example, a video file may contain a video stream, an audio stream, and optionally other streams like subtitle or data streams.

- **Frames**: The individual images that make up a video stream. A video file is essentially a sequence of frames that are displayed rapidly to create the illusion of motion. In FFmpeg, frames are decompressed packages meaning they can contain data additional to images such as sound and timestamps.

The AVFrame datastructure is the frame produced when decoding and decompressing video data with FFmpeg. The AVFrame data structure is a core component of the FFmpeg library, serving as the primary structure for representing decoded video and audio data. This structure is used extensively in FFmpeg to manage and manipulate raw media data.

An AVFrame structure represents a single frame of either audio or video. For video, an AVFrame holds one image, and for audio, it contains multiple samples instead. The structure contains a wealth of fields for describing the frame's properties, such as its format (pixel format for video or sample format for audio), width and height (for video), number of channels and samples (for audio), and other associated metadata.

The data field of an AVFrame is an array of pointers, pointing to the actual image or sound data. For video, the data is typically organized as a two-dimensional array of pixel values, with each row of pixels corresponding to a line in the image. For audio, the data is typically a one-dimensional array of audio samples.

The structure of AVFrame is crucial for our application as we want to transfer only the data that is altered between frames in order to create an efficient PCI-Express data transfer. It can be visualized using Figure 2.4 below:

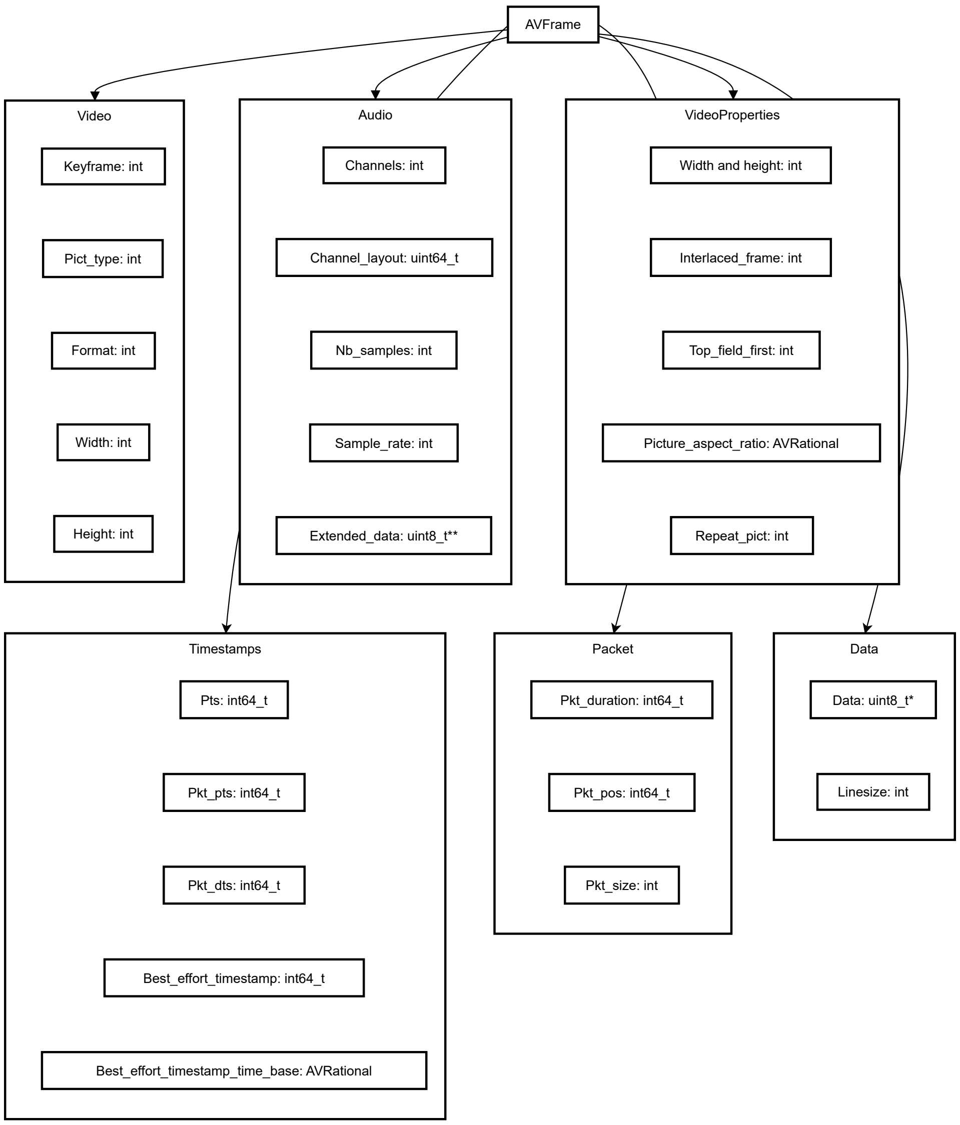


Figure 2.4: Illustration of the AVFrame ata structure from the FFMPEG library created with mermaid. The positioning has been altered in order to increase readability.

Looking at how the AVFrame data structure is built up, a main takeaway is that the raw pixel data is not stored within the data structure itself but rather as a pointer to a data storage address. This is relevant for a few reasons when it comes to data transfer. The *data* pointer contains the raw image data. It's size depends on the format, size and general metadata within the AVFrame structure. From this, we can observe that knowing the static context information is necessary to determine the data size per frame but also sufficient to rebuild the data frame after a transfer.

2.9 Visual framework: SDL

In order to display the video processed by FFmpeg and transferred with SISCI, we need some sort of framework for an application since C does not come with GUI capabilities out of the box. In the realm of media processing, the Simple DirectMedia Layer (SDL) library is often utilized to display raw image data. SDL provides a structure known as an `SDL_Texture`, which is essentially an efficient, driver-specific representation of pixel data. To display raw image data using SDL, one first needs to convert the image data into an SDL-compatible format, subsequently creating an `SDL_Texture` that can be rendered onto the screen.

Initially, raw image data, such as that found in an AVFrame structure from FFmpeg, is typically in a format not directly compatible with SDL. Thus, it needs to be converted to a format SDL can understand, such as SDL's own pixel format. For cross-platform purposes it is important that the framework chosen is compatible with a wide range of operating systems, including Windows and common distributions of Linux.

2.10 Compiling for Dolphin product compatibility

Compilation of SISCI projects is most easily done using a makefile. The SISCI developers kit comes with libraries under `/opt/DIS/include` and `/opt/DIS/src/`. These are essential for the linker in order to create an executable that can utilize SISCI functionality.

In order to create the *.o* object files, the include paths have to be passed to the GCC compiler. In order to create executable files, the linker needs to be passed the argument ”`LFLAGS=-Wl,-L/opt/DIS/(LIBDIRECTORY),-rpath, /opt/DIS/(LIBDIRECTORY),-lsisci`” at the end of the argument list. The specifics of creating a makefile that supports the entire program will vary based on hardware and which configuration the software is desired to work in. The makefile used for this project is found in Appendix C.

3 Specification

3.1 Functional specification

I order to fit the relatively open problem description provided by Dolphin, the system needs to provide support for several different modes, inputs and outputs. The system should be able to:

- Process video files of different formats, sizes and lengths into packets ("frames") that are transferable through a PCI-Express cluster and reconstructable into the original format.
- In the case where there is no file but instead raw data coming from PCI-Express compliant camera hardware, the data should be able to be handled the same way a processed frame would by the cluster network.
- Transfer video data from one machine to another using PCI-Express and direct cabling.
- Multicast video from one machine to several nodes that receive at the same time.
- Support real-time video playing on one or multiple devices receiving the video data frame-by frame over PCI-Express.

3.2 Performance requirements

Although there is inherent value in simply showing that the PCI-Express technology is versatile enough to support a wider range of appliances, there should also be some performance-based backing as to why that is advantageous. The application should meet the following performance requirements:

- The application should be able to handle dimensions big enough to require large frame sizes without compromising on frame rate or other metrics.

- The application should support clients or server turning off or being disconnected during transfer without crashing the program.
- Theoretical transfer speeds (disregarding video processing) should be comparable to Ethernet - based alternatives and ideally outperform them. PCI-Express transfer speeds as determined by generation and cabling width is specified in section 2.2. This applies to a theoretical node with a fast enough graphical unit to where the bottleneck lies in the transfer as opposed to the processing of the video.

3.3 Interface requirements

This application is intended as an open-source demonstration tool for the services provided by Dolphins products. As such, it should be operateable and understandable by anyone who is interested in this type of technology.

- The program should be easy to extend to other hardware or software requirements through modular interfaces.
- The program should be easy to run and modify to the users specific needs.
- The application should be designed and implemented with future expansions in mind.

4

Design

4.1 Hardware design

The hardware design is determined by the gear supplied for the project by Dolphin. They supplied three computers with installed PXH830 adapter cards, as well as a MXS824 switch with four PCI-Express 320-10060 cables. In order to meet the requirements, the nodes have to be interconnected and at the same time leave room for extensions in the form of an external GPU or a PCI-Express compliant camera.

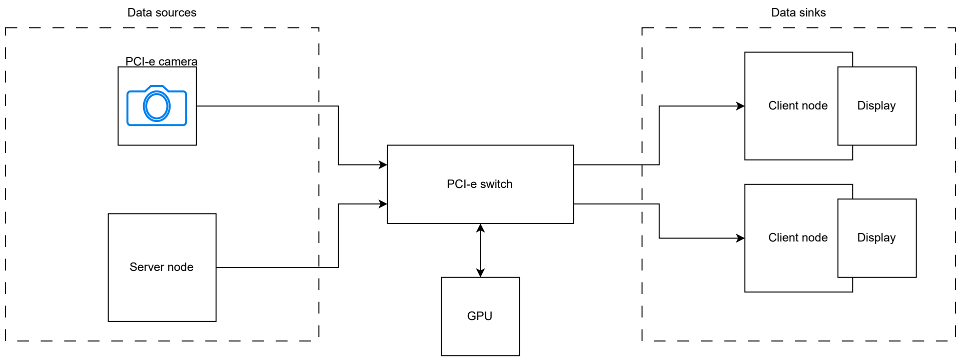


Figure 4.1: Figure: A hardware design overview that includes all potential application extensions including GPU and camera support.

Since the hardware design is determined mostly by what is available for development in terms of Dolphin products, the software will have to be designed around the hardware. The way to design around several potential application expansions is modular design.

4.2 Modular software design

The software can be designed in a modular way to allow the different pieces of the hardware system to access relevant and necessary functionality. Modularization in this code brings several key benefits, especially considering the project's potential to evolve in various directions. By breaking down the code into distinct modules, we increase the system's adaptability to deal with the different possible extensions. It also means we can add to or change individual modules without affecting the whole system. This is particularly useful for this project, as there are several possible enhancements but some uncertainty as to which ones there will be time to implement. With a modular approach, we can develop without having to overhaul large parts of the code. This not only streamlines our development process but also ensures we can quickly adapt to new requirements or opportunities, making the most of the project's potential. The implementation of modularization for the specific purposes of this application is explained further in section 5.8.

The flow of data through the modules for the different modes is illustrated in Figure 4.3, Figure 4.2 and Figure 4.4 below. Through splitting the program into different modules, the goal is to be able to switch hardware modes seamlessly. This should provide compromiseless to the performance requirements.

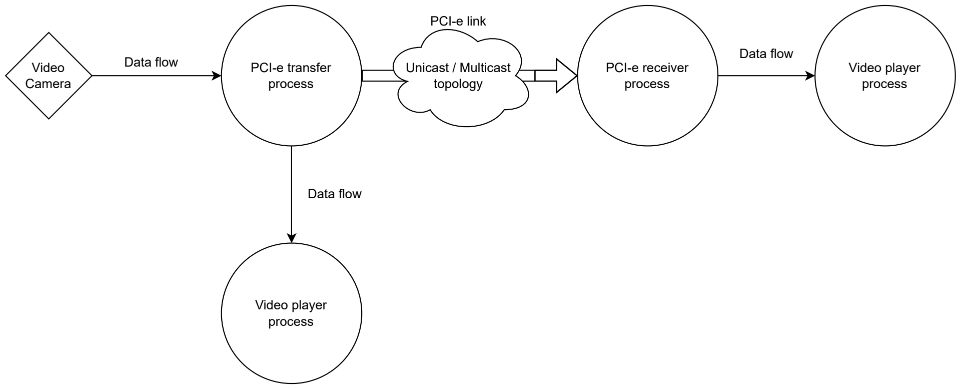


Figure 4.2: Figure: How data flows in the case of a camera being the source of the video frames. The circles represent running modules where the thin lines are one-to-one data transfers while the thick arrows represent a potential multicasting of the data to several instances of the receiving module.

A single module should provide the interface needed to make a decoded video file appear no different from a direct frame video stream from a camera. The next image illustrates how simply adding a module should provide sufficient. One thing to keep in mind is that the feed of frames into the PCI-Express process will depend on the video decoding capabilities of the system and potentially create a bottleneck. This is as opposed to the system in Figure 4.2 where the feed of frames would be decided by the frames per second capture rate of the camera.
The GPU processing module should be standalone in a way that makes the clients displaying the video unaffected as to whether a GPU is used in the cluster or not.

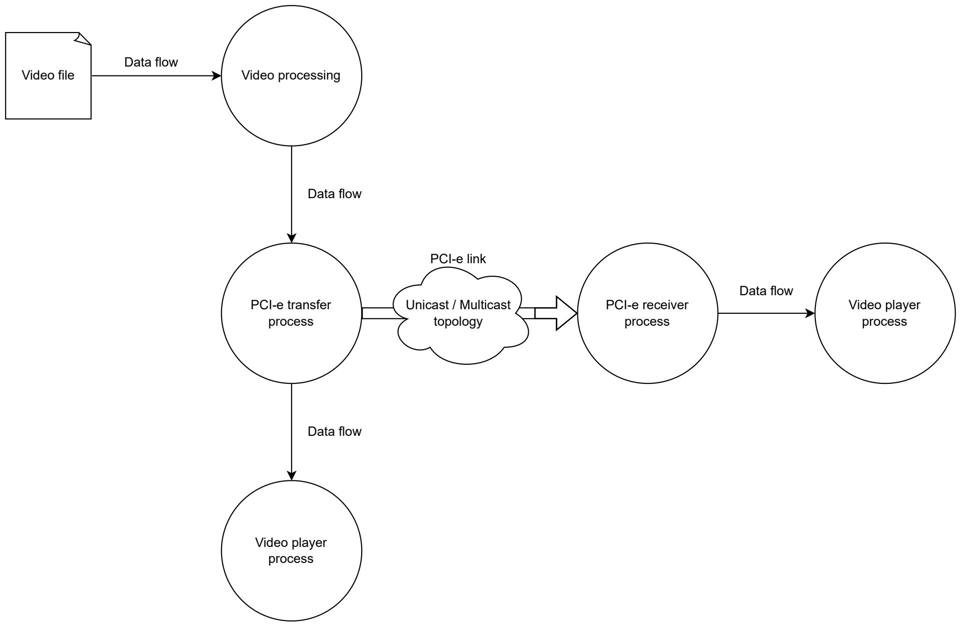


Figure 4.3: Figure: How data flows in the case of either previous source of video frames with a GPU unit attached to the cluster network for video processing. The circles represent running modules where the thin lines are one-to-one data transfers while the thick arrows represent a potential multicasting of the data to several instances of the receiving module.

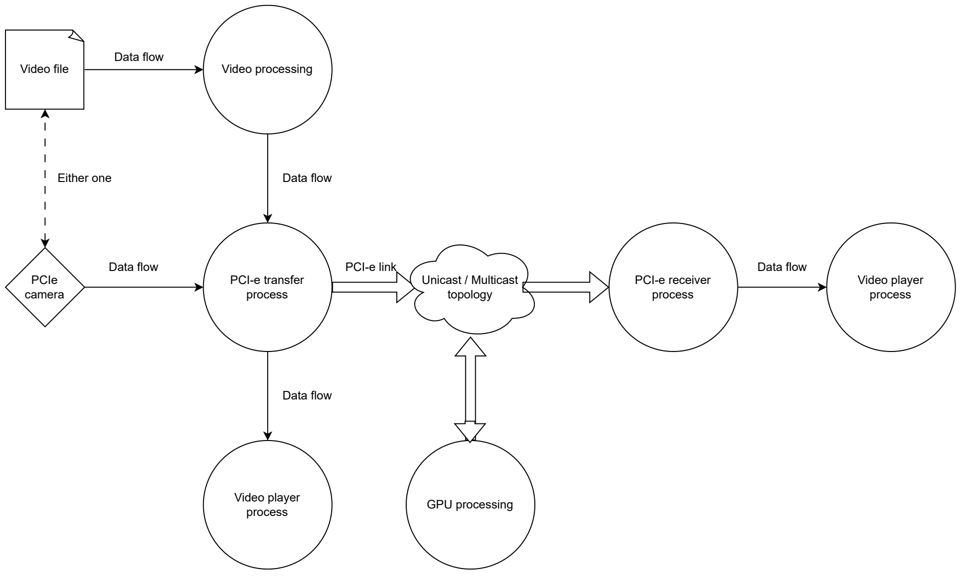


Figure 4.4: Figure: A hardware design overview that includes all potential application extensions including GPU and camera support.

5 Implementation

5.1 Overview of implementation sequence

Due to opting for a rewrite of existing demo capabilities, the implementation sequence was as follows:

1. Develop the (existing) unicast capability on a physical cluster consisting of two nodes, interconnected by Dolphin hardware.
2. Develop the same functionality but only using multicast transfers. Multicast works like unicast in a two-node cluster.
3. Expand the hardware with a PCI-Express switch and a third node, then add support for this in the software.
4. Add a GPU to the cluster connected to the switch and develop software for it to be able to participate in the cluster.

5.2 Physical cluster setup (Without multicast support)

The PCIe cluster was set up using two desktop computers running fresh installations of Ubuntu 18.04.01. Both nodes were equipped with PXH830 adapter cards provided by Dolphin. The computers were connected using four cables of the type 320-10060 which provides the maximum 16x PCI-Express physical link between the PXH830 adapters. The setup is illustrated below in Figure 5.1.

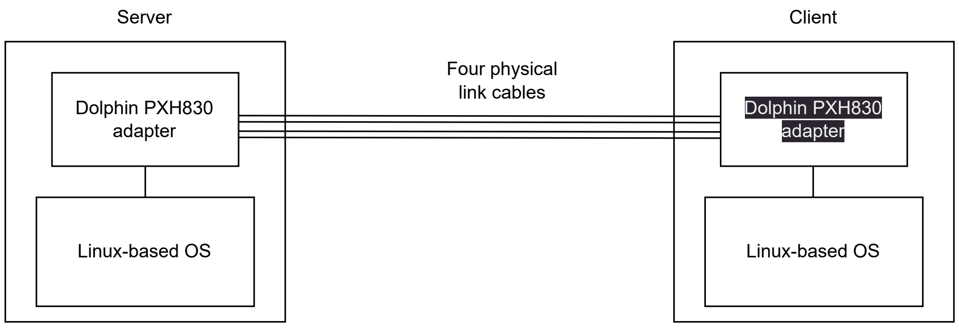


Figure 5.1: Simple illustration of the hardware setup for unicast.

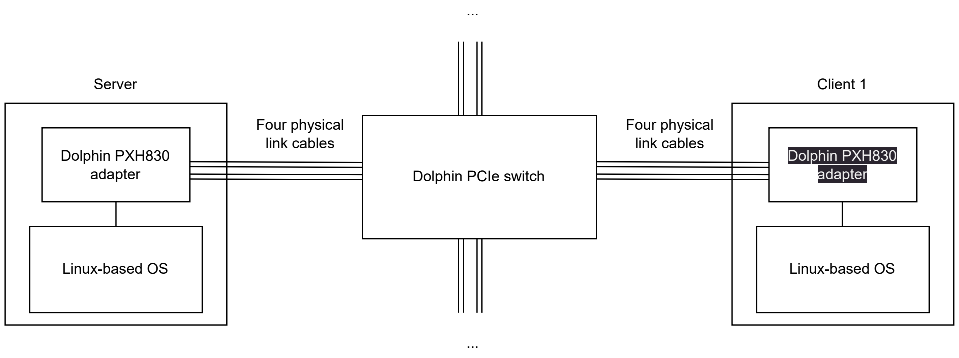


Figure 5.2: Simple illustration of the hardware setup for multicast.

5.3 Physical cluster setup (Multicast)

The physical difference between the unicast and the multicast setup is simply the inclusion of an MXS 824 Dolphin PCI-Express switch as well as more nodes with their corresponding physical link cables. The inclusion of the switch allows easy connections of other devices to the cluster such as GPUs but the cluster still needs reconfiguring in order to make sure every node has a separate and known node ID. Figure 5.2 illustrates the multicast setup without an included GPU but with two free sets of cables where a GPU connect is possible, mirroring the physical setup.

5.4 Node configuration

The SISCI software stack is compatible with most common operating systems and is delivered with a self-installing archive (”SIA”) that installs required drivers and libraries for our PCI-Express functionality. Quite a few options had to be tried in order to make this work but eventually success was found with Ubuntu 18.04.01 as the operating system on all nodes. Dolphin responded to the feedback regarding some of the broken SIA packages and claim to have fixed them but that has not been verified during this development process. In general, software built for older systems tend to be more stable.

The SISCI software stack installs for the entire cluster and can be monitored from a ”frontend manager node”. The manager node can be one of the cluster nodes running SISCI-based software but it can also be a different machine. For remote testing purposes, the cluster manager suite was installed on a personal laptop allowing tests to be run from remote locations. The cluster manager node gets a set of scripts that allow GUI monitoring of the installed nodes in the cluster. This gives information about the status of the cluster but also access to pretty powerful features like rebooting nodes and getting node IDs present in the cluster. In order to get this access, the SIA (for the correct O.S.) has to be run from the management node with Ethernet access to all nodes that shall be configured in the cluster. This means setting up SSH access to all nodes.

Since the cluster management node in this case did not run Ubuntu 18.04.01, the install process for every time the cluster changed (including adding a switch for multicast) was a heterogeneous one which is performed as follows:

1. Remove the current configuration on all nodes by running SIA with flag –wipe
2. Run the SIA on all nodes using the flag –install-node
3. Run the SIA on the frontend manager node with the flag –install-frontend
4. Verify cluster configuration using the script ”DIS_Admin” on the frontend node.

This procedure does not mention setting up SSH access to the nodes for the frontend install to configure correctly. There are several ways of doing this, this setup utilized the package ”openssh-server”.

5.5 Modular program setup in C

5.5.1 Multithreading

The video processing module operates using a custom queue data structure called `AVFrame_Q` which as the name suggests, is a queue of AVFrames. The idea behind using this is that it is hard to tell what takes more time between breaking down frames using the ffmpeg library or transferring them using SISCI and PCI-Express. This could depend on the graphical capabilities of the node running the server, the type of Dolphin hardware we have installed as well as other loads affecting both the node and the cluster network. To mitigate this, we place the processed frames into a queue that supplies atomic push and pop operations as well as other helper functions that can be seen in Figure 5.3. This lets us run the video processing functionality in a different thread from the one handling the PCI-Express transfer.

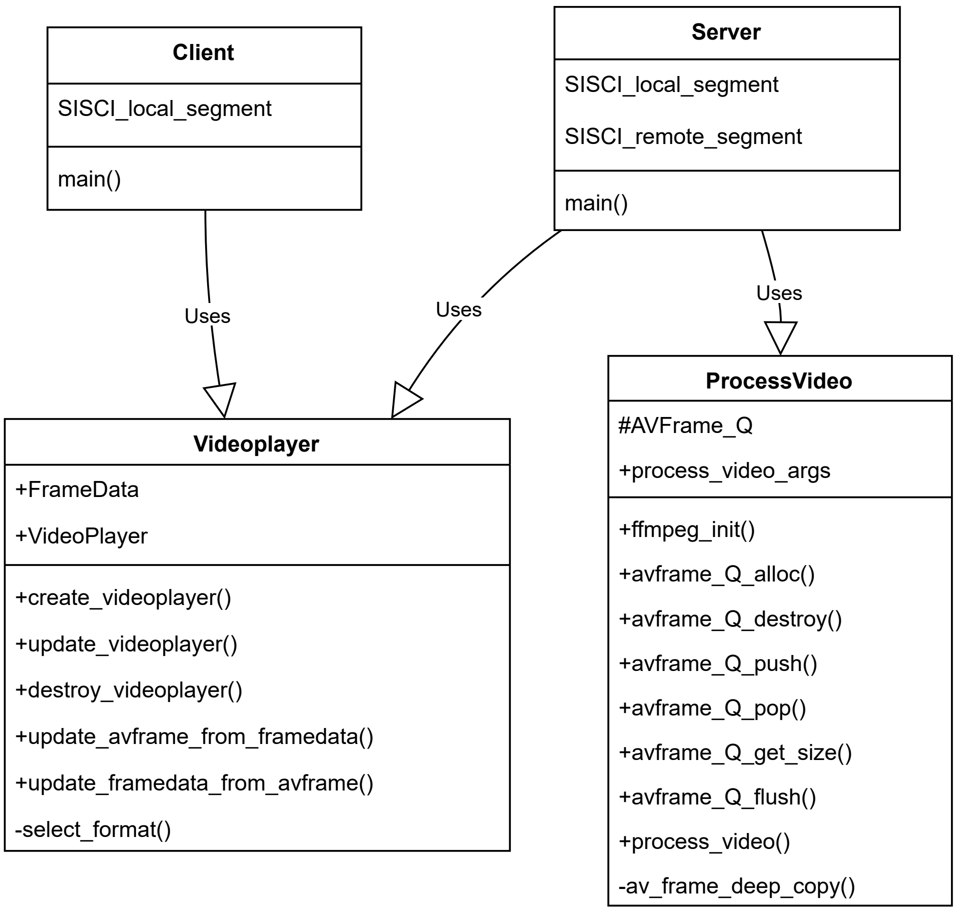


Figure 5.3: A Class-diagram-like approach to illustrating the modular interface. ”+” Indicates that the data/function is accessible when importing the module. ”-” Indicates that it is a non-accessible function for internal use only. ”#” Indicates that the data structure is opaque and can be referred to by the importing module. The internal data is hidden for the importing module however, meaning that only pointers to the imported datatype is allowed.

The two threads deal with filling up and emptying the queue separately which gives us a buffer in case one of the processes experience a delay.

The interaction is showed in the following schematic:

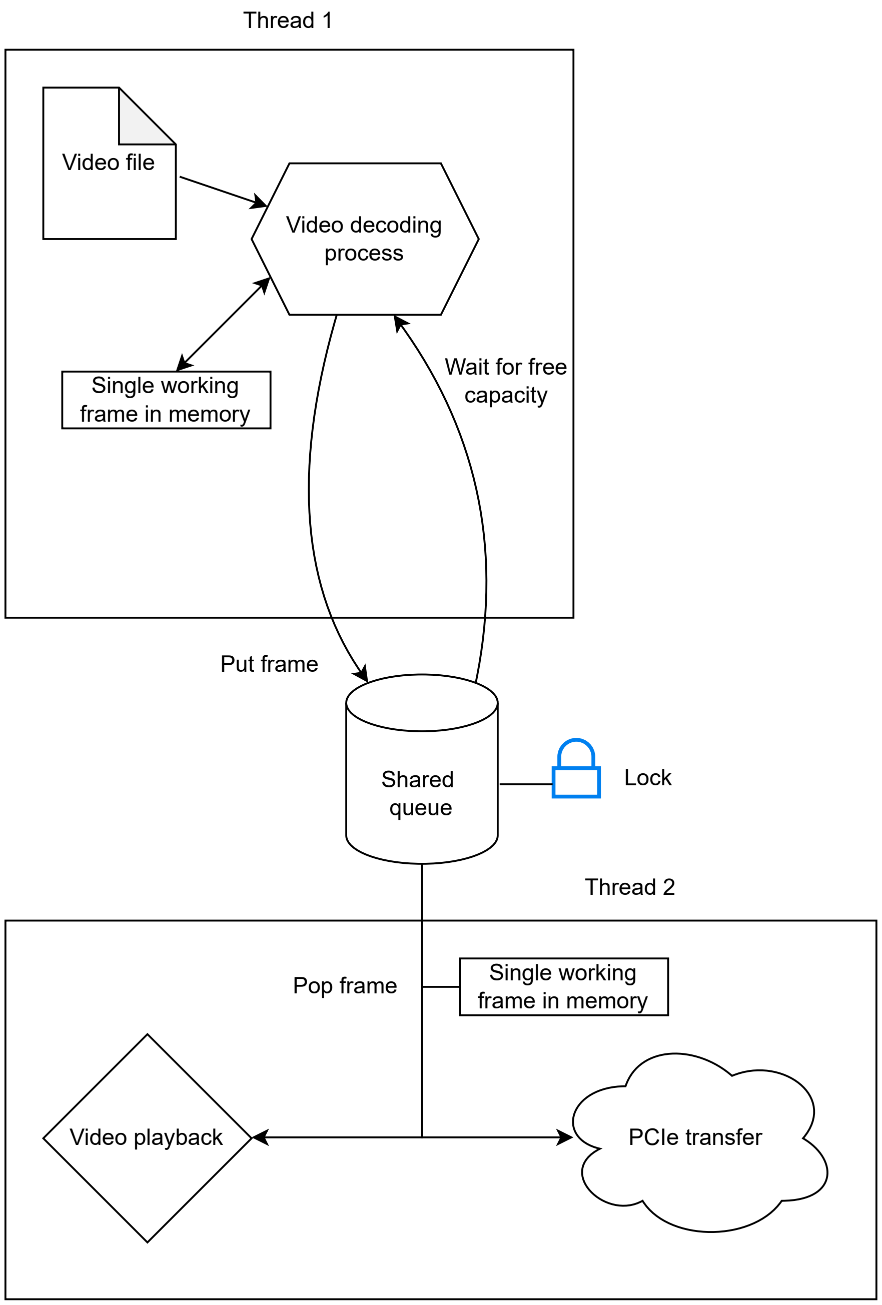


Figure 5.4: Illustration of how the server node uses two threads to interact with a custom queue data structure.

The queue is protected by a mutex and semaphores to ensure it works as a safe connection between the running threads.

5.6 Video processing

The video processing process is the one illustrated as ”thread 1” in Figure 5.4. It also supplies the queue datastructure and an interface with all functions needed to use the queue from an importing module. The interface is made as a `.h` file and is found in teh appenidix’ subsection D.

The module processes a video file into a single allocated `AVFrame` structure which functions as a ”working frame” and uses the provided `AVFrame_Q_push()` function to add it to the shared queue before processing the next frame. The data is deep copied into the queue and not passed as a reference. We avoid passing pointers which is the ”normal” way of handling AVFrames because they themselves contain a lot of dynamically allocated variables. This was done to avoid confusing pointer arithmetrics in a multithreaded environment.

First, the ffmpeg library is initialized by calling `av_register_all()` and `avcodec_register_all()`. This sets up the necessary environment for video processing. Next, the input video file is opened using `avformat_open_input()`. This function establishes a connection to the video file and retrieves the format-specific information into an AVFormatContext structure. Once the format is established, the stream information is extracted by calling `avformat_find_stream_info()`. This step allows us to identify the video stream within the file.

To decode the video, the appropriate codec is determined using `av_find_best_stream()`. The codec information is stored in the codec variable. A codec context is allocated and initialized with `avcodec_alloc_context3()`. This context holds the parameters and settings required for video decoding. The codec parameters from the input stream are copied to the codec context using `avcodec_parameters_to_context()`. This ensures that the codec context is properly configured for decoding the video. The codec is then opened using `avcodec_open2()`. This step prepares the codec for video decoding.

To hold the decoded video frames, an AVFrame structure is allocated using `av_frame_alloc()`. This frame will store the uncompressed video data. Then begins the decoding and processing loop. The function reads packets from the video file using `av_read_frame()`. If the packet belongs to the video stream, it is decoded using `avcodec_decode_video2()`. If decoding is successful and a frame is obtained, it is processed further. The processed frame is then pushed into a queue, enabling further processing or display. This is achieved by calling `avframe_Q_push()`. The loop continues until a predefined threshold is reached or the video ends. If the video ends, the function seeks back using `av_seek_frame()` to ensure continuous playback. After the loop exits, the allocated resources, including the frame, codec context, and format context, are properly freed to release memory.

In summary, this module initializes the necessary structures, opens the video file, decodes and processes video frames, and ensures proper cleanup. The process is based almost entirely on ffmpeg functionality and is perhaps better understood by studying the code or interface of the module.

5.7 PCI-Express transfer

The SISCI library is initialized using `SCIInitalize()`. This SISCI library function finds an optimized copy function by checking the system CPU and and installed SISCI adapter and is important for data transfer later. It also does simple error checking by checking that the driver version is consistent with the SISCI library version. In addition, system resources needed for SISCI memory transfer later is allocated.

For every synchronizing segment planned to use in the application, a separate virtual device has to be created. A virtual device is created with `SCIOpen()` and requires a reference to a descriptor of the virtual device with the datatype `sci_desc_t`. The virtual device is a communication channel to the SISCI driver and uses a descriptor to ensure accessibility while keeping the exact implementation details behind an opaque handle.

Once a communication channel is established with the driver, is becomes possible to open a SISCI segment using `SCICreateSegment()`. This function allocates contiguous memory of the specified size with a segment ID for identification. The SISCI driver will keep a list of allocated segments that can be referred to and used in our program by passing their respective segment ID.

From here, the allocated segment can be prepared for use with other machines in the cluster by making it accessible to the Dolphin adapter. This is done with `SCIPrepareSegment()`. Since it is possible to have multiple adapters installed, teh function needs to be passed the correct adapter ID and segment ID in order to link them up properly.

At this point, the segment memory is allocated in kernel space and managed by the SISCI driver. The idea is to map the memory (which synchronises using the SISCI driver) to the user space program as a virtual address. The `SCIMapLocalSegment()` function returns a virtual address that the program can write to and read from using regular memory operations (such as *memcpy()*). The SISCI driver handles synchronization letting us operate on the memory address as if it were regular local memory while affecting memory present physically elsewhere.

The segment can then be prepared for remote access from another device running the SISCI software stack. Notably, we prepare segments **for remote access by another computer** in the client software. That way, the server is prepared to write directly to the client instead of the client fetching data from the server. This can be thought of as a mailbox operation where the address of the mailbox is mapped directly into the address space of the server program. To acomplish this, the segment is set available using

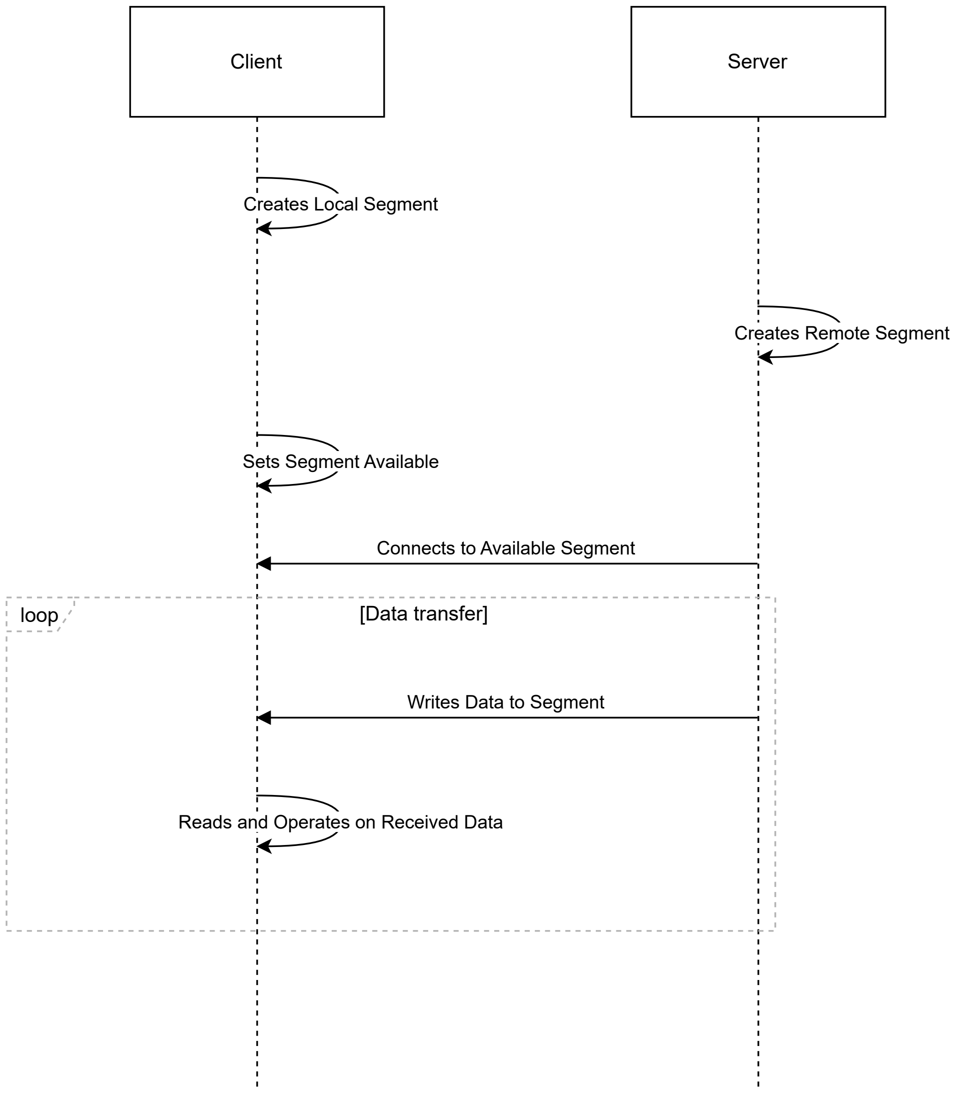


Figure 5.5: Sequence Diagram: Segment preparation. The mapping of remote and local segments into program address space has been omitted.

SCISetSegmentAvailable() on the client side followed by SCIConnectSegment() on the server side. Since SCIConnectSegment() only succeeds if the segment has already been set available, we can opt to run it in a loop as a blocking call. This ensures that program features such as video transfer only proceeds once a client is actually connected. The video server program runs the following loop:

```
do {
        SCIConnectSegment(v_dev_framedata,
                          &r_seg_framedata,
                          DIS_BROADCAST_NODEID_GROUP_ALL,
                          FRAMEDATA_SEGMENT_ID,
                          ADAPTER_NO,
                          NO_CALLBACK,
                          NULL,
                          SCI_INFINITE_TIMEOUT,
                          SCI_FLAG_BROADCAST,
                          &err);

        SleepMilliseconds(10);

    } while (err != SCI_ERR_OK);
```

This ensures there is a client available before the video processing starts.

Once the segments have been properly set up for communication, data transfer can be done simply by writing to the mapped memory sequence. The choice is given between using regular processor I/O operations on the mapped memory or DMA transfer protocols provided by the Dolphin adapter cards. For PIO operations, SISCI provides the function SCIMemCpy() which provides an optimized memory copy algorithm based on hardware surveys performed when SCIInitialize() was ran. For bigger data transfers, reducing CPU load by using provided DMA capabilities might be a better option. In order to use the DMA capabilities present on the Dolphin adapter cards, a DMA queue in local memory can be created using SCICreateDMAQueue(). The data can then be transferred from local memory to a remote segment using SCIEnqueueDMATransfer() and SCIPostDMAQueue(). This requires more overhead so it is only worth it if reducing processor load is of importance and the data transfer is big enough to take advantage of the DMA capabilities.

After data transfer has been completed in a SISCI program, several cleanup operations should be performed to ensure the system resources are properly released. Firstly, the SCIUnmapSegment() function should be called to unmap any previously mapped segments from the application's address space. This includes both local and remote segments that were mapped. Secondly, ff any segments were made available to remote nodes using the SCISetSegmentAvailable() function, they should be made unavailable using the SCISetSegmentUnavailable() function. Any segments that were created using the SCICreateSegment() function should be removed using the SCIRemoveSegment() function. Finally, the SCIClose() function should be called to close any virtual devices that were opened with the SCIOpen() function. After all these operations, the `SCITerminate()` function should be called to terminate the SISCI API. This ensures that all resources allocated during the use of the API are properly released. A simplified illustration of a segment setup sequence is seen in Figure 5.5.

5.7.1 Differences between unicast and multicast segment setup

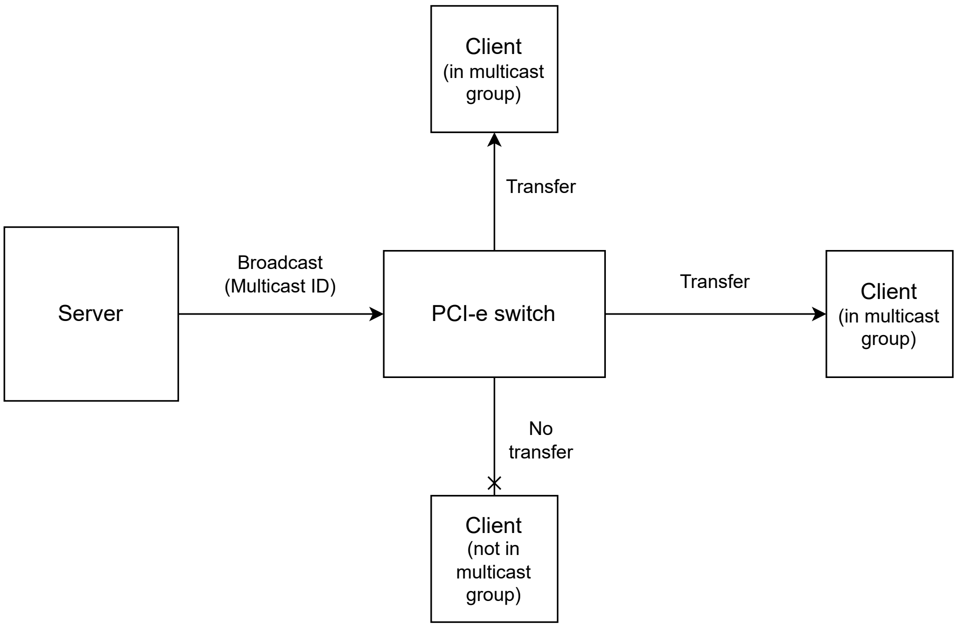


Figure 5.6: Figure: A PCI-Express switch handles the package distribution to nodes meaning there are no longer any direct connections.

In order to set the application up for multicast, there are a few changes required to the PCI-Express transferring setup. Multicast is not a direct memory transfer between two nodes which means the previous setup where a client makes a segment available for the server to connect to has to be altered. Instead, the packets are distributed to the nodes with segments registered in the corresponding multicast ID group as illustrated by Figure 5.6. This makes the segment setup different but also enables hot-plugging of both server and nodes. The nodes can be set up to ”listen” to a multicast segment and play once data comes through and the server is able to send data to the switch regardless of whether there are nodes receiving or not.

Both server and client nodes create and prepare segments using `SCICreateSegment()` and `SCIPrepareSegment()` respectively, passing the flag `SCI_FLAG_BROADCAST` and a multicast segment ID to prepare for multicast compatibility. The segment is then mapped to local memory using `SCIMapLocalSegment()` and set available similarly to how the client would prepare a segment previously. This mapped address serves as the read address. As opposed to previously, both server and client create local segments. Additionally, the segments share the same segment ID on all participating nodes. The server also needs to connect to the multicast segment using `SCIConnectSegment()` and then map it again, this time as a **remote** address using `SCIMapRemoteSegment()` to create a **write** address. This method of creating local segments on all nodes but only mapping a remote write address for the server allows there to always be a segment to write to no matter how many clients are connected. This gives us the benefit of being able to reboot both server and clients without interfering with the program.

5.7.2 Differences in what data to transfer

Since the unicast implementation needs an established and alive connection between server and client, the size of the data segment should be established before data transfer ensues. As previously mentioned, the raw pixel color data needed to create a displayable frame has a size dependent on the video size and the pixel format. The SISCI segment is allocated first by the client node that imports the data, which means we need a way to communicate the required segment size from the server to the client before the transfer ensues. This is done in the following way:

1. Client creates a segment and waits for a connection.
2. Server connects to the segment and transfers the segment size and other information necessary for video reconstruction based on a test video processing run.
3. Client closes the segment and re-allocates it with the now correct size.
4. Server re-connects.
5. Video processing and transfer begins.

With this unicast implementation the metadata is transferred once at program startup and then the connection re-establishes for continuous frame data transfer. In the multicast implementation, several other factors have to be considered. Serving an unknown number of client nodes means we can not assume the clients all have the necessary metadata when we transfer frames. The way this is solved is by establishing two different multicast segments. One for metadata and one for raw pixel data. The client listens to the metadata segment and makes sure all necessary variables are initialized before starting to process the data available on the raw data segment. This double-segment method of data transfer not only ensures multiple client (multicast) functionality but also has the added benefit of allowing both clients and server to reboot without major issues to the program. Advantages and disadvantages will be examined in further detail in chapter 8.

5.8 Video display

One goal of development was to create the video display functionality as a modular design since the functionality for viewing the video should be the same on both the client and the server side of the operation. The modular design should allow the user of the module to view it as a simple black-box that takes an AVFrame and displays video in return, illustrated in Figure 5.7.

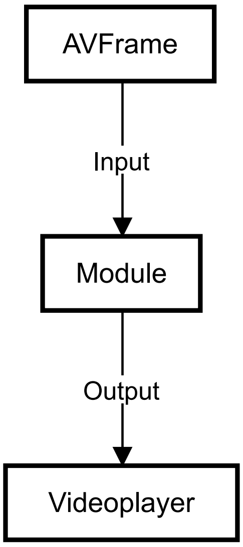


Figure 5.7: Black-box logic for reusability of the module. An AVFrame should be sent in and displayed on the screen for the required amount of time.

There are multiple reasons to modularize code. In this instance, the reusability aspect of having the same video playback mechanism in both server and client nodes is an obvious one. Reusability when the code expands to support more functionality is also equally important. Other benefits of modularization include:

- **Ease of Understanding**: Modular programming breaks down complex systems into smaller, manageable parts, each with a specific functionality. This makes the code easier to understand and learn.
- **Maintainability**: In a modular program, each module can be updated, modified, or fixed independently without affecting other parts of the program. This makes maintaining the program easier and reduces the chance of introducing new bugs when making changes.
- **Encapsulation**: Modular programming promotes encapsulation by keeping the internal workings of each module hidden from the others. This helps to prevent data corruption and unauthorized access.

The video player module was set up in the following way: The function `create_videoplayer()` is responsible for initializing SDL, creating an SDL window and renderer, and setting the color for clearing the renderer. If any of these operations fail, the function will report an error and terminate the SDL subsystem.

Once the video player has been initialized, the image shown on screen can be updated simply by calling `update_videoplayer()` and passing the new AVFrame that should be rendered on screen. This function receives an AVFrame and a VideoPlayer structure as input and translates the AVFrame into an SDL texture. The texture, containing the video frame data, is then updated and rendered onto the screen. A delay is introduced to maintain the desired frame rate. Once the frame is displayed, the SDL texture is destroyed to manage resources effectively.

Inconveniently, ffmpeg and SLD use different datatypes to store pixel formats so we use a custom function to select the correct SDL pixel format based on the ffmpeg format. The function `select_format()` handles the conversion. This step is vital because it ensures that the video frame data decoded by FFmpeg is translatable into a format that SDL can render. This method is however, not guaranteed to work for every file format which means the function has to be extended based on the required functionality.

SDL is a good choice for an application build on SISCI as both project are built to be platform independent. SDL serves as an abstraction layer between the hardware and the application, handling low-level tasks such as rendering graphics, capturing input, and playing audio. This means that when using SDL, the developer doesn't need to write different code for different operating systems or hardware configurations. Instead, SDL automatically translates the application's commands into instructions that the specific hardware can understand. This capability not only saves considerable development time and effort but also improves code maintainability and scalability. In addition, SDL is optimized for performance and supports many advanced features, making it a robust choice for multimedia applications that need to run on diverse hardware setups.

The interface of the video display module can be found in the appendix' subsection E.

5.9 Adding a GPU to the cluster.

After successfully implementing the multicast software, one of the open ports on the PCI-Express switch was used to expand the cluster with a Nvidia Xavier GPU. The Xavier was pre-configured with SISCI drivers from Dolphin and had yet another PXH 830 adapter card attached to it's PCI-Express slot.

The GPU was configured to read frames from the same multicast address as the running cluster. Then, instead of rendering graphics and showing them, the GPU was configured to flip the frame upside-down and send the flipped frame over a segment with a new multicast ID. Since the PXH830 supports four multicast IDs, segment ID 0 was used for the frame metadata, ID 1 for the raw frame data and ID 2 for the flipped frame data. The flow of data between segments inside the modules is illustrated below in Figure 5.8.

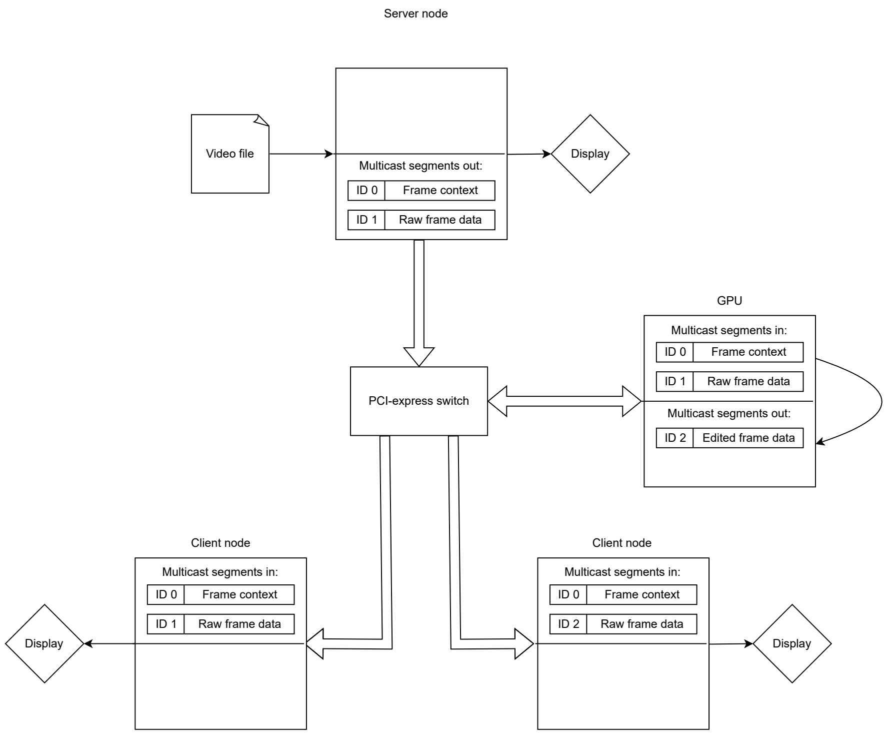


Figure 5.8: Illustration of how the different nodes participating in the cluster utilize the segment IDs (0-2). Here, the GPU transformation represents whatever transformation is desired.

The following snippet was used to flip the frames:

```
void flip_frame(AVFrame *frame) {
    struct SwsContext *sws_ctx;
    AVFrame *flipped_frame;
    int ret;

    // Allocate a new frame
    flipped_frame = av_frame_alloc();
    if (!flipped_frame) {
        fprintf(stderr, "Failed to allocate frame.\n");
        return;
    }

    // Set the frame data
    flipped_frame->format = frame->format;
    flipped_frame->width = frame->width;
    flipped_frame->height = frame->height;

    // Allocate the frame data
    ret = av_image_alloc(flipped_frame->data, flipped_frame->
    linesize, flipped_frame->width, flipped_frame->height,
    flipped_frame->format, 32);
    if (ret < 0) {
        fprintf(stderr, "Failed to allocate frame data.\n");
        av_frame_free(&flipped_frame);
        return;
    }

    // Create a scaling context
    sws_ctx = sws_getContext(frame->width, frame->height, frame->
    format, frame->width, frame->height, frame->format,
    SWS_BITEXACT, NULL, NULL, NULL);
    if (!sws_ctx) {
        fprintf(stderr, "Failed to create scaling context.\n");
        av_freep(&flipped_frame->data[0]);
        av_frame_free(&flipped_frame);
        return;
    }

    // Flip the frame
    flipped_frame->data[0] += flipped_frame->linesize[0] * (frame
    ->height - 1);
    flipped_frame->linesize[0] = -flipped_frame->linesize[0];

    // Convert the image
    sws_scale(sws_ctx, (const uint8_t * const*)frame->data, frame
    ->linesize, 0, frame->height, flipped_frame->data,
    flipped_frame->linesize);

    // Free the scaling context
    sws_freeContext(sws_ctx);

    // Copy the flipped frame data to the original frame
    av_frame_unref(frame);
    av_frame_move_ref(frame, flipped_frame);

    // Free the flipped frame
    av_frame_free(&flipped_frame);
}
```

A client node was then set up to render the flipped graphics from segment ID 2 by simply switching which segment ID it uses for it's local segment. This works because the transformation is as simple as flipping the image and does not affect the metadata. The client uses the metadata from the server and the frame data from the GPU to render it's frames. In the case of a more advanced image processing technique, the last multicast segment ID could be set up to carry the altered metadata information in order for the client software to still support the video. This was not attempted in the current implementation.

6 Testing

Testing the installed cluster and software is a crucial part of this process for several reasons.

- Setting up the cluster was a long and error-prone process as referenced in the appendix' subsection B. Verifying that the cluster and physical links were operating as expected is therefore a prerequisite for software development to be useful at all.
- Testing the multicast performance is required to identify hazards when moving from a unicast to a multicast transfer mode.
- Lastly, testing the application to ensure it meets the specification is a requirement.

6.1 Environment

All tests were run directly on the deployed cluster which consists of the system described in chapter 5. This introduces the potential issue of hardware differences which will be discussed further in chapter 8.

Two of the nodes including the one consistently running the server side of the application had the following specs:

Component	Description
System Name	HP Compaq 8200 Elite SFF PC (XL510AV)
Processor	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
Total Memory Size	8GiB System Memory
PCIe Bridge	Xeon E3-1200/2nd Gen. Core Processor Family PCI Express Root Port

Table 6.1: System Specifications

The last node had the specs found in Table 6.2.

Component	**Description**
System Name	OptiPlex 7010 (OptiPlex 7010)
Processor	Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz
Total Memory Size	4GiB System Memory
PCIe Bridge	Xeon E3-1200 v2/3rd Gen Core processor PCI Express Root Port

Table 6.2: Hardware Information

The nodes each had installed a PXH830 card from Dolphin. Additionally, each PHX830 card was connected to a central Dolphin MXS824 switch with four PCI-Express 320-10060 cables from Dolphin. This gives a PCI-Express 3.0 x16 link although the PCI-Express bridge on the motherboards limited the actual transfer speed to x8.

6.2 Validation of cluster configuration

SISCI conveniently comes with a suite of testing software in order to validate the hardware setup. Notably we can validate that the cluster works as expected both for direct PIO transfers and reflective multicast transfers with the tests `scibench2` and `reflective_bench` respectively. These benchmarks are part of the SISCI demo library which can be build by running:

```
make -f Makefile.demo all
```

in the `/opt/DIS/src` folder that is created upon installing the SISCI developers kit.

Figure 6.1 shows the result of the throughput benchmark `scibench2`.

The results show that the physical link is running at x8 capacity since any less would downgrade the throughput to a maximum of 4000Mb/s. The offset from the theoretical throughput of 8000MB/s can partially be explained by the MPS as explained in subsection 2.2.1. Diagnosing the cluster using `dis_diag`, a SISCI developer tool that comes with the frontend install, we see that the MPS of the cluster is 128bytes. Using this information we can adjust the expected data throughput to:

$$8000\,\text{MB/s} \times \left(\frac{128 - 20\,\text{Bytes}}{128\,\text{Bytes}} \right) = 6750\text{MB/s} \tag{6.1}$$

There is still some loss of throughput unaccounted for but because the performance is sufficiently close to the theoretical one and the expected performance of `scibench2` is unspecified, the cluster can be assumed correctly configured.

`reflective_bench` provides benchmarking of the multicast cluster setup shown in Figure 5.2. Running the benchmark is no different from running `scibench2` except two nodes run the benchmark in client mode and one as server.

A typical run of the benchmark can be seen in Figure 6.2.

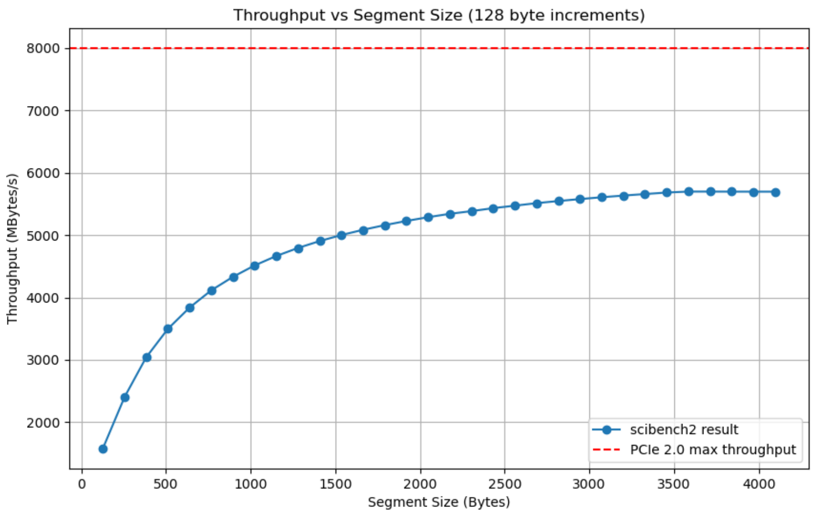


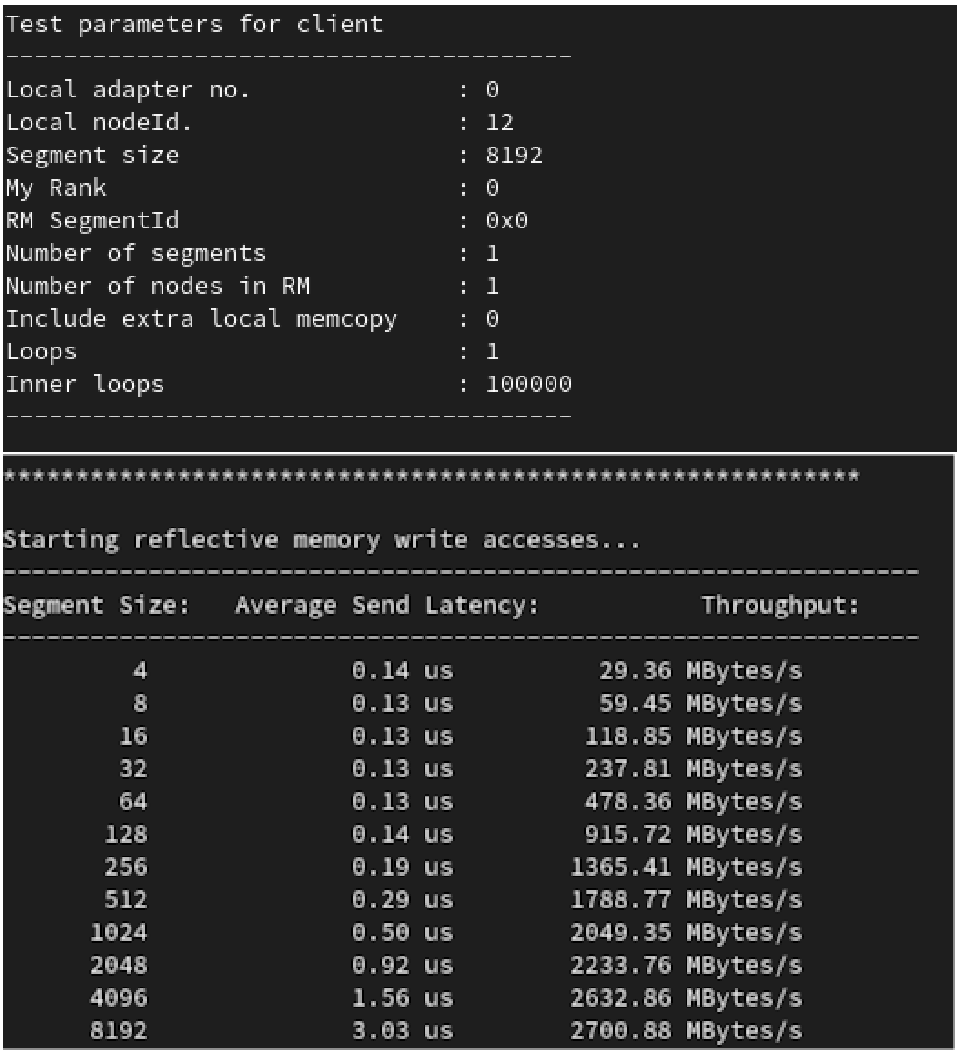
Figure 6.1: A plotting of the results from running the benchmark `scibench2`. The benchmark was configured to return the average of 10000 segment write loops for each segment size. The sizes benchmarked were 128-4096 byte segments with 128 byte increments. For comparison the red line represents the theoretical PCI-Express 2.0 throughtput maximum.

We can see that the throughput is about halved compared to `scibench2` which is to be expected because this benchmark measures traffic both ways compared to just in one direction.

6.3 Validation of video transfer compatibility

For validation of video compatibility, the test library found at Anonymous (2023) was used as a source for test files. All files found here were able to be transferred over the multicast system successfully. However, the larger dimension video files trigger an error when allocating a multicast segment for the uncompressed frame data saying it is unable to get system resources. To fix this issue for large dimensions and through extension frames, the option for `dis_max_segment_size_megabytes` in the SISCI driver has to be increased from the default value of $4MB$. This is specified in Dolphin (2016). The maximum allowed multicast segment size is $2GB$ but by simply adjusting the driver to $8MB$, the bigger files in the test library worked as well.

It is possible to require a certain file format when transferring from file, and this was the behavior of the previous application Dolphin disposed for unicast demonstration. However, the spec requires certain file compatibility with common file formats and is set up to handle several kinds of raw image frame data. This is to allow for future camera support, being that a camera would provide a raw image data stream in an unknown format.



```
Test parameters for client
---------------------------------------
Local adapter no.               : 0
Local nodeId.                   : 12
Segment size                    : 8192
My Rank                         : 0
RM SegmentId                    : 0x0
Number of segments              : 1
Number of nodes in RM           : 1
Include extra local memcopy     : 0
Loops                           : 1
Inner loops                     : 100000
---------------------------------------
```

```
************************************************************

Starting reflective memory write accesses...
----------------------------------------------------------------
Segment Size:   Average Send Latency:          Throughput:
----------------------------------------------------------------
         4              0.14 us         29.36 MBytes/s
         8              0.13 us         59.45 MBytes/s
        16              0.13 us        118.85 MBytes/s
        32              0.13 us        237.81 MBytes/s
        64              0.13 us        478.36 MBytes/s
       128              0.14 us        915.72 MBytes/s
       256              0.19 us       1365.41 MBytes/s
       512              0.29 us       1788.77 MBytes/s
      1024              0.50 us       2049.35 MBytes/s
      2048              0.92 us       2233.76 MBytes/s
      4096              1.56 us       2632.86 MBytes/s
      8192              3.03 us       2700.88 MBytes/s
```

Figure 6.2: Figure: An output example of `reflective_bench`. The benchmark returned the average of 10000 segment write loops for each presented segment size.

The longest consecutive running of video transfer was a 16 hour session with a single video looping. This was done to validate that a memory leak previously present in the `AVFrame_Q` structure was fixed.

6.4 Simultaneous video stream testing.

Simultaneous testing of the multicast application can be done simply by running the program as in two different processes in each machine. Because the multicast `SEGMENT_ID` is limited to $0-3$ and the application uses one `SEGMENT_ID` both for the metadata transfer and the actual frame data transfer, the maximum possible running instances of the application was limited to two. Setting the ID's to $0,1$ for the first instance and $2,3$ for the second one is all that is needed for two simultaneous video multicasts.

6.5 High-performance throughput testing.

In the theoretical situation where the video decoding hardware is able to decode frames faster than they can be transferred over PCI-Express, the multicast transfer module would become the limiting factor. It can be shown that this is not the case for the current hardware by inspecting the size of the `AVFrame_queue` during application runtime and see that it is almost always empty. In order to test the actual capabilities of this module without a bottleneck coming from the video decoding or texture rendering modules, the following test was implemented:

1. Modify the program to not use the video player functionality. This means no updating of the SDL video texture upon retrieval of a new frame on either the server or the client.
2. Fill the `AVFrame_queue` with frames before starting the main transfer loop instead of alongside the main transfer loop. The queue can be initialized to a fitting capacity.
3. Measure the time between the first and last frame arriving at the client.
4. Use the uncompressed frame size, number of frames and time measurement to calculate the throughput independent of the video decoding and playback modules.

The modified module structure to accomplish this is shown in Figure 6.3.

When testing, the average of ten runs transferring 100 pre-processed frames of the size 345600Bytes over multicast, the results were as follows:

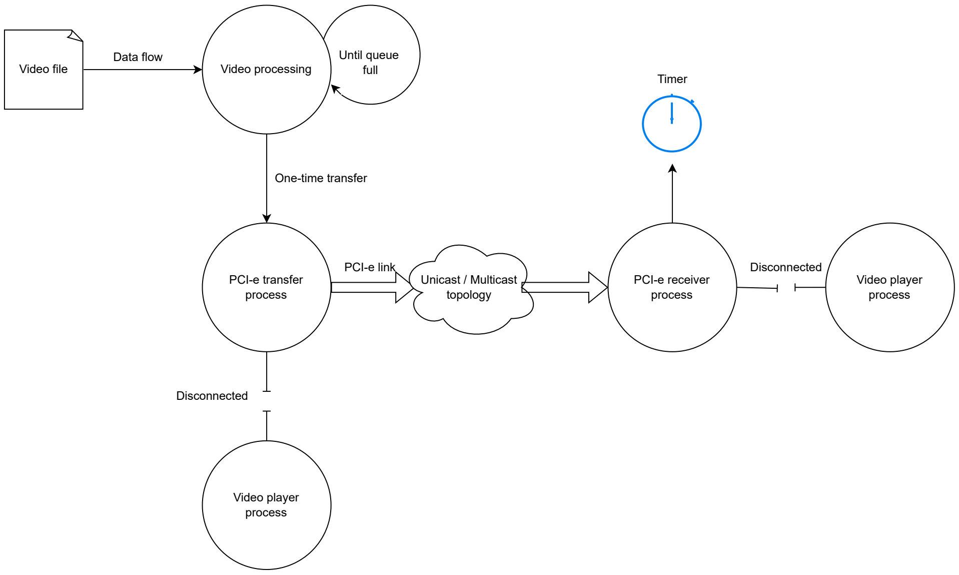


Figure 6.3: Figure: Modifications to enable the measurement of throughput.

Transfer method	**Average transfer time**	**Throughput**
PIO	0.102160	338 MB/s
DMA	0.764302	452 MB/s

Table 6.3: Transfer times

Notably, this is quite a lot lower than the speed measured in the previous test. However, it is still higher than the maximum throughput available through the most common ethernet standards 1000BASE-TX and 10GBASE-TX as presented in Table 2.2. The result of this test will be examined further in chapter 8.

7

Results

7.1 Recap

The previous sections detailed the implementation of an application written in C that is able to play video simultaneously on several computers using Dolphin's SISCI multicast functionality. The hardware implementation of this was done using custom Dolphin adapter cards in regular desktop computers interconnected by a PCI-Express switch. The software was designed in a modular fashion to accommodate expansions and used ffmpeg for the video breakdown module, SDL for the video display module and Dolphin's SISCI API for the multicast transfer module.

7.2 Product

The main product of this thesis is a first-of-its-kind open-source video transfer application that utilizes SISCI multicast to play video on several nodes at once. The application is able to:

- Transfer and play video in real-time on all connected nodes using PCI-Express and SISCI multicast.
- Break a video file down into frames that can be transferred over PCI-Express. The application supports several common video formats and dimensions of video files.
- Allow fairly high performance on limited hardware which should scale alongside improved hardware.
- Support hot-plugging of both clients and server, allowing mid-transfer reboots and joining of new clients.

The application demonstrates features provided by Dolphin's SISCI API such as multicast in a way that did not exist previous to this paper. In addition, it lays the foundation for

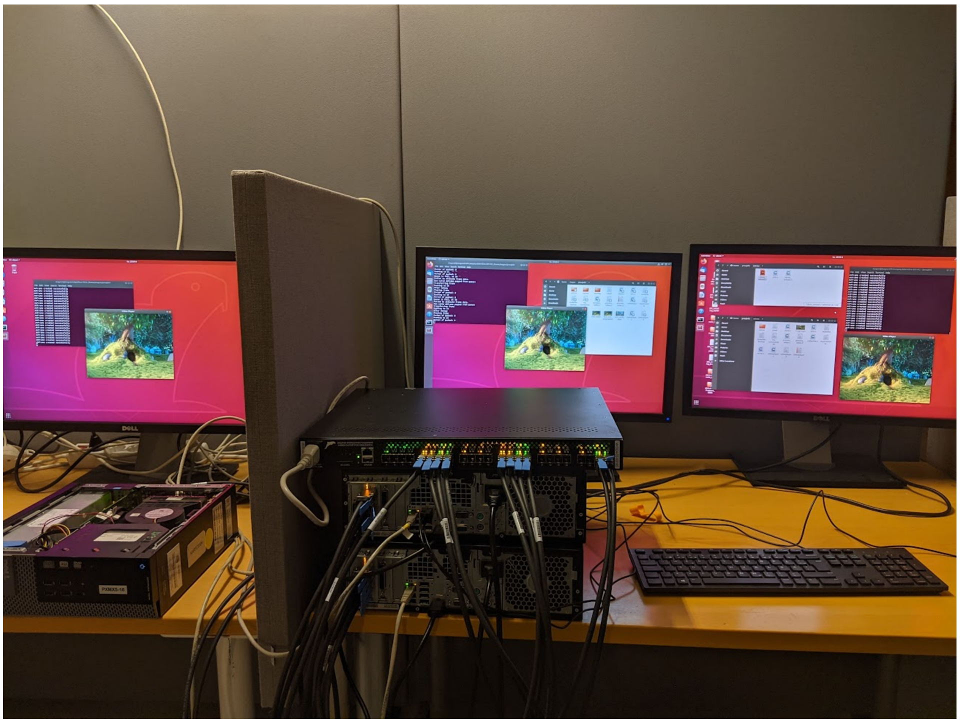


Figure 7.1: Picture showcasing multicast of a video file from one server to two clients. The three nodes are each connected to the PCI-Express switch and a display.

expansion through sending the video stream through an external GPU for processing, allowing for demanding computational processing to be done outside of the server and client nodes. The future potential of this will be discussed in chapter 8.

The full, functioning code used for the multicast system can be found at Angvik Hovdar (2023). An image displaying the running setup is found above in Figure 7.1.

The secondary product is the contents of this paper which describes the design, implementation and testing scheme used to create this system as well as the progress made on how expansions would fit into the larger system. This also includes compromises and negative results which would serve helpful to anyone looking to re-build, improve or expand on any part of this project.

7.3 Accessibility improvements

This paper serves to make the technology researched and deployed more accessible for academic purposes as well as professional. The design philosophy and open-source nature of the project makes the application well suited for expansions as well as re-use in other projects with overlapping interests. Through building a modular, open source demo application, the application is made accessible for reuse and reconfiguration to fit other experiments within and outside the scope of this exercise.

Additionally, the future expansions of the demo capabilities have been made more accessible. The proposed ideas of using a camera to directly supply video data over PCI-Express as well as advanced GPU processing of real-time data are closer to realization after the research done in this project.

7.4 Challenges and solutions

One major challenge with using multicast for video frame transfer is that the segments are created in static sizes independently of each other. This is as opposed to an imported ”regular” SISCI segment where the size of the segment is decided by the node that creates it and imported by the other.

Given that the size of the segments may vary depending on the format and dimensions of the video, this information needs to be synchronized somehow. This is to avoid the video being interpreted wrong on the receiving end in the case of changes on the server-end.

The application solves this by using **two** multicast segments. A set-size one with metadata and a variable size one that is re-created with a new size in the case of metadata changes. This is a fully functional way to solve this problem with both advantages and disadvantages that will be discussed further in chapter 8.

8 Discussion

8.1 Hardware limitations

The hardware used for this task limits the potential of the technology in two main ways:

1. The video decoding capabilities of the hardware (A regular desktop computer with no GPU was used)

2. The PCI-Express port on the motherboard of said computers only supports PCI-Express generation 2.0. In other words, the throughput of the system will always be bottlenecked by this port no matter which PCI-Express generation or configuration.

The testing results would not be accurate in the case of hardware upgrades or other differences. Upgrading the PCI-Express limiting factor of the 2.0 ports on the motherboard would make the whole system see improved transfer speeds. Similarly, any improvement to the video processing capabilities of the nodes would affect performance as well.

Still, the hardware used for the purposes of this development process was plenty in the sense that it was never necessary to make any special considerations. It is more so the fact that an application running on older specs is anticipated to run even better on newer specs that is important. The hardware and software stack from Dolphin supporting the application should allow this assumption to hold in this scenario as well.

8.2 Choices regarding development

8.2.1 Re-building for open-source

The choice to re-develop an application to make it open source compared to expanding the existing application has both advantages and disadvantages.

On the one hand, re-developing the application for open source can lead to increased transparency and involvement. It allows for a potentially larger pool of developers to adapt and contribute to PCI-Express technology. It also encourages the use of open standards and interoperability which is desired by Dolphin as well.

On the other hand, building on the existing application would have been a lot faster as many of the features that took time developing were already solved problems. There is also a risk of starting from scratch in introducing new bugs or compatibility issues. However, building on the existing application would not offer the same level of transparency as an open-source solution.

8.2.2 On setting up a cluster from scratch

Configuring the physical cluster for use before starting development on the software application took a lot longer than anticipated. This was the case although the cluster topology was more or less decided by the available hardware. Through the expansion of the pre-configured GPU it became evident how much time would have been saved by using a pre-configured hardware supplied by Dolphin.

Although the experience regarding how SISCI drivers are configured is useful for someone seeking thorough knowledge of the complete Dolphin stack, the actual software and demo showcase development was hindered by this approach. Additionally, not having any control over which driver installation packages were up to date for the various operating systems lead to problems that could only be solved by developers external to the task at hand.

Ultimately, there will always be a trade-off between how desired the knowledge of the entire system is and how much time should be spent tackling already solved problems. In the case of an equivalent problem, using the exact setup used in this paper or collaborating more closely with the hardware supplier on this part of the design process would probably lead to an improved result.

8.2.3 Choosing development direction

The problem description presents three different directions in which to take the development. The implemented solution focuses on completing the multicast part of the problem with an additional proof-of-concept for the GPU. This was in part because a camera was unavailable during the period where developing for it was an option but also because a proper multicast implementation would enable future development on the other problems.

A major discussion point when deciding which direction to go was finding out whether the different expansions were limited by other factors. For instance, using a camera which interface would only enable writing to a file would not be any different from using the regular server operation. However, writing the raw data directly to a multicast address would mean not having to process the video which is a big difference in how the system works. While more interesting, this approach would require significant time and research on the camera and that was sadly not an option at the time.

Having an external GPU in the cluster allows processing to be done mid-network outside of the nodes responsible for showing the video. While just a proof-of-concept that could just as well have been done by the nodes was performed in this instance, the options for expanding this is more interesting. However, advanced video processing is a difficult field on its own and due to lack of specific competence, it was regarded as more worthwhile to spend time building a reusable multicast base that enables this as a future expansion.

8.2.4 Data flow through the system

Figure 5.4 illustrates how both threads use a ”working frame” as a buffer both as a location to push to queue from and destination for the popped frame. Practically, this means that the frame data switches memory location a total of three times before actually being multicast by the adapter to the client nodes. This is not the most efficient implementation, but it does provide transparency, modularity and ease of expansion in terms of which data to transfer. Perhaps this was an approach that affected the tested theoretical transfer speed negatively.

The data flow between nodes is separated into a set-size multicast segment for metadata and a variable size multicast segment for raw data where the size is determined my the metadata information. Seeing as multicast segment ID’s are a very limited resource (0-3 for the PXH830), this wastes some potential in the application. Some considered options are:

- Using a ”regular” segment to communicate the context information to the clients. This would mean losing the ability to hot-plug clients and servers automatically which is a useful multicast feature. To mitigate this, software can be built to keep track of clients active in the system and re-establish connections to them. Ultimately, building a unicast framework to ensure the multicast functionality works as intended seems somewhat obsolete.

- Combining the two segments’ contents into one segment with a variable size but a fixed minimum size. Accomplishing this means adding more advanced methods of reading from segments and separating out the data. Additionally, synchronization mechanisms for adjusting to new formats and dimensions would have to be built in a more complicated way. The upside to this is using half the segment ID’s which is a pretty big improvement. Transferring all data in one go could also have a positive effect on transfer speed. Out of the two, this is probably the better option as there are no downsides except for increased complexity.

8.3 Test result discussion

8.3.1 Theoretical application throughput

While the test results presented in Table 6.3 were sufficient for the current video transfer application, it is still less than 10% of the theoretical transfer speeds from the testing in Figure 6.1. This is an area that could be improved upon to allow for very high quality video transfer at solid frame rates. Some possible improvements to achieve this are:

- In order to get a good measurement of throughput, a video file with large dimensions resulting in large frames is preferred. The segment size is initiated to the size of a video frame and a multicast segment has a maximum size of $2GB$ according to Dolphin (2017a). In other words, since segment size isn't a limiting factor, larger files could improve performance.
- The current program writes to two segments each frame cycle in order to be able to alert the client of any potential metadata changes. Putting this into one segment is possible but requires some finesse in how the segment size is initiated (and probably reinitiated). It would be more efficient to transfer all data in one operation, especially if using a DMA.
- There are several DMA optimizations not implemented in this design or testing scheme. Dolphin (2020) specifies among other things DMA queueing and asynchronous waiting mechanisms as options that could improve the throughput of the demo application.

Taking a look at high quality video, we can for instance see that for uncompressed $4K$ $60FPS$ video with an assumed color depth of $8bit$ for three channels (RGB), the required data rate is:

$$\begin{aligned}\text{Resolution} &= 3840 \times 2160\,\text{pixels} = 8294400\,\text{pixels/frame}\\ \text{Data per frame} &= 8294400\,\text{pixels/frame} \times 24\,\text{bits/pixel} = 199065600\,\text{bits/frame}\\ \text{Data rate} &= 199065600\,\text{bits/frame} \times 60\,\text{frames/second} = 11943936000\,\text{bits/second}\\ \text{Data rate in MB/s} &= \frac{11943936000}{8 \times 1024 \times 1024} \approx 1425\,\text{MB/s}\end{aligned} \tag{8.1}$$

The current implementation which is limited by PCI-Express 2.0 would actually support the maximum of two simultaneous $4K$ $60FPS$ video streams since two multicast segments are used per instance out of a maximum of four. This shows the potential for performance without even improving the hardware.

8.4 Further work

As pointed out in the task description, the total workload of the ideas presented exceeded what was possible to complete during the span of a masters thesis. This section will therefore present the opportunities for future work as well as relevant findings about the remaining tasks that have been discovered or researched during this thesis.

8.4.1 Camera

The concept of this application can be extended to have a camera write directly into the memory of the server application for raw frame data to be distributed and played the same way it is when coming from a current file in this implementation. Connecting a camera directly to the PCI-Express cluster without going through the server node is also an interesting option but will depend on the camera in question. Thorough research on PCI-Express compliant cameras is needed to make this an integration. Some things to look into are:

- Developing a driver for a PCI-Express compliant camera to be able to use multicast segments directly.
- Sending the data to a GPU unit in some way and letting the GPU handle multicast to the nodes in a similar way to how it was showcased in this project.
- Using PCI-Express to write directly into a buffer on the node used for this application. The server could work as a middle-man to get the camera data out into the multicast protocol.

8.4.2 GPU processing

The GPU showcased in this project was used to flip frames which is not something a GPU would be needed for. Integrating actual CUDA or other graphical programming frameworks to apply transformations that the nodes would spend too long doing themselves would be a good end-goal for this. In particular, running object detection could be a useful feature for a cluster similar to this one sending camera source data to several monitors.

8.4.3 Performance increase

This paper has identified possible increases in performance specified in subsection 8.3.1 which would allow extremely high quality video transfers even on PCI-Express 2.0 hardware. This would be more feasible on a system with nodes that have the GPU power necessary to produce 4K frames at the demanded speed or in a system that has managed to interconnect a very high quality camera.

Optimizing the data flow and theoretical application throughput as discussed in the sections above could lead to exciting improvements. Especially if combined with a high-performance task that can take advantage of the optimizations such as high-quality or multiple parallel video transfer.

9 Conclusion

This project's implementation of a video sharing application using SISCI multicast provided a working application with extra thought put into extensions and future possibilities. The resulting product is built from the bottom up and provides availability of the SISCI multicast feature set through a design and implementation that is meant to be expanded upon as well as conceptual use of extensions using a GPU.

Certain difficult decisions had to be made during the development, and some ideas had to get priority over others. This means there is still further research needed both into unfinished parts of this problem and related opportunities that have been found during this project.

Finally, this thesis hopes to spark and inspire further work within the field of PCI-Express technology. This application breaks new ground with its multicast video sharing features but it is still merely an introduction to what can be done using the same principles. It is the hope of all contributing parts that the results and reflections provided in this paper would serve helpful to anyone wishing to look further into this technology.

Bibliography

Angvik Hovdar, M., 2023. Project code repository. URL: https://github.com/22mah22/prosjekt.

Anonymous, 2023. Home | Test Videos. URL: https://test-videos.co.uk/.

Cubedo, S.A., 2021. Fast Multi-GPU communication over PCI Express URL: https://www.duo.uio.no/bitstream/handle/10852/88547/sivertac-thesis.pdf?sequence=1.

Dolphin, 2015. Dolphin Express - Remote Peer to Peer made easy URL: https://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_IX_Peer_to_Peer_whitepaper.pdf.

Dolphin, I., 2016. Dolphin eXpressWare Installation and Reference Guide. URL: https://www.dolphinics.com/download/CW_SOFTWARE/OPEN_DOC/DIS_Install_Guide_5_4_0_CW_Linux.pdf.

Dolphin, I., 2017a. PCI Express Reflective Memory / Multicast URL: https://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_reflective_memory.pdf.

Dolphin, I., 2017b. SISCI Users Guide URL: https://www.dolphinics.com/download/SISCI/OPEN_DOC/SISCI_API_2_users_guide.pdf.

Dolphin, I., 2019. PXH830_users_guide.pdf. URL: https://www.dolphinics.com/download/PX/OPEN_DOC/PXH830_users_guide.pdf.

Dolphin, I., 2020. SISCI_api_2_functional_specification.pdf. URL: https://www.dolphinics.com/download/SISCI/OPEN_DOC/SISCI_API_2_functional_specification.pdf.

Dolphin, I., 2023. PCI Express Gen3 switch | MXS824 | Microsemi. URL: https://www.dolphinics.com/products/MXS824.html.

IEEE, 2022. IEEE 802.3:2022, IEEE Standard for Local and metropolitan area networks - Media access control (MAC) and physical layer specifications. doi:10.1109/IEEESTD.2022.3146033.

Lye, T.A., 2010. A PCI Express communication interface for DMP camera arrays URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/252276/382972_FULLTEXT01.pdf?sequence=1.

PCI-SIG, 2006. PCI Express® Base Specification Revision 2.0. URL: https://members.pcisig.com/wg/PCI-SIG/document/download/8246.

PCI-SIG, 2010. PCI Express® Base Specification Revision 3.0. URL: http://www.pcisig.com/specifications/pciexpress/base3/.

Skrede, S., 2022. MPI Over PCI Express Networks URL: https://www.duo.uio.no/bitstream/handle/10852/96939/Master_Thesis___Jakob_Skrede_and_Sigurd_Sonne_.pdf?sequence=11&isAllowed=y.

Appendix

A Problem description

Master thesis: Advanced Video transfers over PCIe

Version 1:

Background

Dolphin is designing and manufacturing advanced PCIe hardware and software products. The products are used world-wide in various medical, defense, automotive, HPC, real-time and control systems and simulators.

Motivation

Dolphin has a basic demo program that can transfer data over PCIe from a PC with a video camera attached to a remote PC for display. Dolphin would like to extend this demo program to better demonstrate some sophisticated PCIe techniques.

Proposal

The work consists of several independent extensions / tasks – It is currently assumed that all of the suggestions will be more work than what can be expected from a 6-month maser thesis.

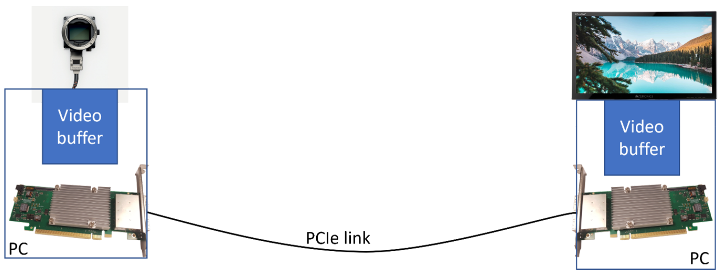
Preferred Background

The master student should have good understanding of PC architectures, real-time systems and familiar with Linux systems and C programming. Experience in GPGPUs programming (Cuda) is a strong benefit for the task involving Direct GPU transfers. The work will not require any special knowledge about PCIe.

Transfer details existing solution

Data from the video camera is written to a memory buffer in the local PC. Data is copied from the memory video buffer to a remote display buffer using PCIe DMA functions. Dolphins SISCI API is used to set up and configure transfers over PCIe.

An overview of the setup is shown in the graphics below.



Proposed Master Thesis

The demo program can be extended in several ways to explore ways to utilize advanced PCIe transfer methods.

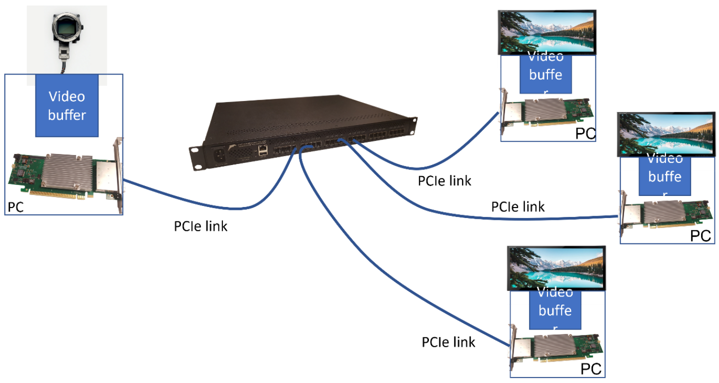
Distribution of video data using PCIe Multicast

Support for PCIe Multicast to distribute the video steam to two or more systems.

Transfer details

Data from the video camera is written to a memory buffer in the local PC. Data is copied from the memory video buffer to a remote display buffer using PCIe DMA functions and PCIe multicast.

An overview of the proposed setup is shown in the graphics below.



Challenges: Understand Dolphins SISCI API and implement efficient data transfer protocols. Complexity level: medium.

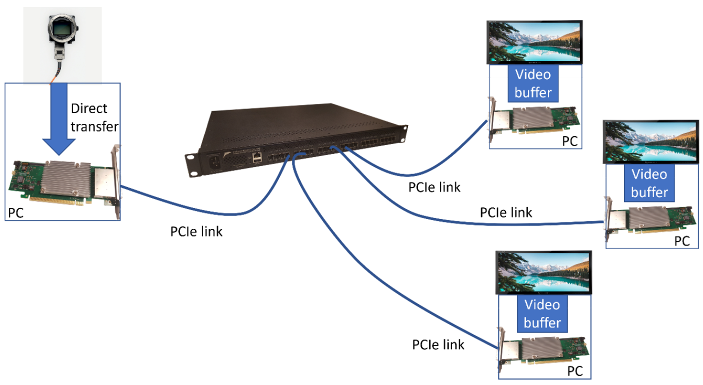
Distribution of video data using Direct Transfers and PCIe Multicast

Support for PCIe Multicast to distribute the video steam to two or more systems.

Transfer details

Data from the video camera is transferred directly from the camera to two or more remote display buffers using PCIe multicast.

An overview of the proposed setup is shown in the graphics below.



Challenges: Modify PCIe camera driver to enable direct transfers over PCIe. To our knowledge, this has never been done by anyone yet. Similar techniques have been demonstrated using FPGAs

Complexity level: high.

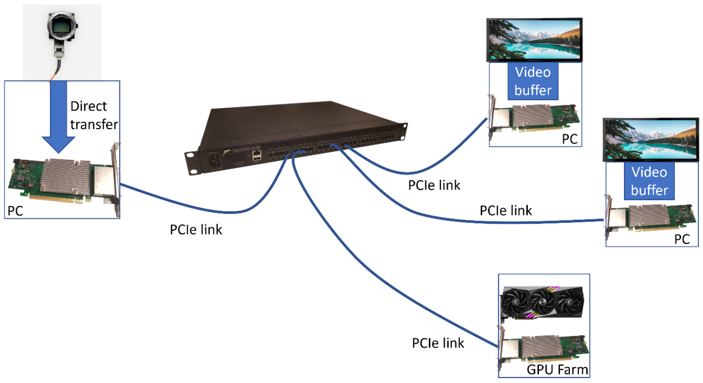
Distribution of video data using Direct Transfers, GPU processing and PCIe Multicast

Video stream is also sent to directly to GPU buffer in a PCIe expansion system. GPU modifies video stream and sends data directly to display buffers in two or more PCs.

Transfer details

Data from the video camera is transferred directly from the camera to two remote display buffers and a GPU buffer using PCIe multicast. The GPU utilizes RDMA to send modified data to two or more display buffers.

An overview of the proposed setup is shown in the graphics below.



Challenges: Understand and set up PCIe multicast directly into GPU buffer. Understand and set up GPU RDMA transfers into PCIe multicast addresses to enable direct transfers into remote display buffers.

Complexity level: high.

More information

More information on Dolphins eXpressWare software, and the SISCI API can be found at https://www.dolphinics.com/products/embedded-sisci-developers-kit.html

https://www.dolphinics.com/solutions/embedded-system-reflective-memory.html

https://www.dolphinics.com/products/dolphin_pci_express_software.html

Genera conditions

The master student will be offered an office at Dolphin's offices in Oslo or can work most/part of the time remotely. If remote work is preferred, the student can borrow the hardware setup. (Some small PCs and required PCIe hardware)

Contact Person

The contact person for this master thesis is Hugo Kohmann – hugo@dolphinics.com. Hugo has several decades of experience in building interconnect solutions, has been mentoring numerous Master thesis and holds a Master degree in computer science from the University of Oslo.

B Note on installation

Note on CISCI installation.

Cisci can be installed using a “self-installing archive” (SIA) provided by Dolphin ICS. The SIA is a script developed for installing all CISCI software required to use Dolpin PCIe hardware such as the PX830 PCIe cards used in my two-machine cluster.
The SIA is available for a variety of operating systems including different versions of Windows and both Debian and Red Hat-based Linux distributions.
A cluster needs at least two nodes that have CISCI installed as well as a frontend for managing and verifying the installation. The frontend can be run on one of the nodes or on a separate machine. I wanted the frontend to be on my personal laptop. As I did not want to install a new operating system on my laptop, I opted for a heterogenous installation as opposed to a homogenous one. In short, this meant I had to run the SIA on each node with the “--install-node” option before running the SIA with the “--install-frontend” option on the frontend computer.
For my installation, I first tried the script for Ubuntu 22.04.01 LTS on a clean install of the operating system on each node. The installation failed because of a missing dependency “libgcc1” which is now called “libgcc-s1”. I could not find a way to circumvent this error and had to abandon Ubuntu 22.04.01. Note: After giving feedback to Dolphin, this error in the script should be fixed although I have not verified it myself.
I then tried clean installs of fedora 36 on my nodes. The SIA failed with a non-descriptive build error and since Fedora 36 was the recommended operating system for this version of the SIA, I eventually abandoned this operating system as well.
My third OS install was a clean version of Ubuntu 18.04.01 LTS on which the SIA worked and I completed the node installations on both nodes.
For the frontend, I was using Fedora 35 and started with the SIA designed for Fedora 25. The script failed with the error returned being ERROR: Failed to build Cluster Management Node packages, #* ERROR: Couldn't build system packages. I then upgraded to Fedora 37 and ran the SIA designed for Fedora 36 which worked.
The final configuration was then: SIA ran with “install-node” for and on Ubuntu 18.04.01 and SIA ran with “install-fronted” for Fedora 36 ran on Fedora 37.
Heads up for frontend installation: The administration GUI tools that the SIA proposes running during installation do not work. Both when trying frontend install on Fedora 37 and on one of the nodes (Ubuntu 18.04.01) the GUI windows would open but be blank, aborting the installation script.
If opted out of, the installation will finish and the GUI tools can be found under /opt/DIS/sbin where they will work. However, as GUI tools only work when run by a non-root user, their functionality is limited. Most Linux distros are not designed for GUI tools to be run by a root user including both Fedora 36 and Ubuntu 18.04.01.

Kjapp rekkefølge:

- Forsøk ubuntu 22.04.1
    - Installasjonsskript stopper på installering av libgcc1 fordi pakken egentlig heter libgcc-s1 og pakkemanager (apt) godtar kun dette.
- Forsøk fedora 36
    - Installasjonsskript returnerer feilkode forbundet med bygging av pakker

- Forsøk ubuntu 18
    - Skript fungerer på nodeinstallasjon ved hjelp av

```
# sh  ./Dolphin_eXpressWare-<version>.sh --install-node --enable-sisci-development --enable-supersockets
```

- Forsøk på cluster node manager installert på ekstern laptop (fedora 35), installasjonsskript for fedora 25.
    - Skript godtar kun tilgang til root@<IP> for nodene, så root bruker må settes opp
    - Skript feiler. “Software finnes ikke på nodene”
- Forsøk på cluster node manager installert på en av nodene

- Installerte på “1 node” via SSH med den andre som cluster manager.
- Kom til å launche graphical tool “dls_netconfig” men den fungerte ikke. Hvit(blankt) vindu
- Hvordan launcher man den nå? Skriptet reboota etter å ha måttet skrive inn passord ca 100 ganger

Heterogenous installation. - Satse på at node install har vært som den skal (og det har den?)

- Fedora 25 på fedora 35 –install-frontend returnerer
- ERROR: Failed to build Cluster Management Node packages, see
- #+ '/home/22mah22/Downloads/DIS_install.log_ByWseO' on 'localhost'.
- #* ERROR: Couldn't build system packages

Fedora 36 da?
Oppgraderte egen pc til fedora 37
fik lov til å bygge system packages nå!
Dis_netconfig (cluster edit) er fortsatt bare et tomt hvitt vindu…
VIndusfeil er Error: BadDrawable (invalid Pixmap or Window parameter) 9
  Major opcode: 62 (X_CopyArea)

- Bare når jeg kjører som root (sudo). Why? Hmm

Kabler i feil vei???

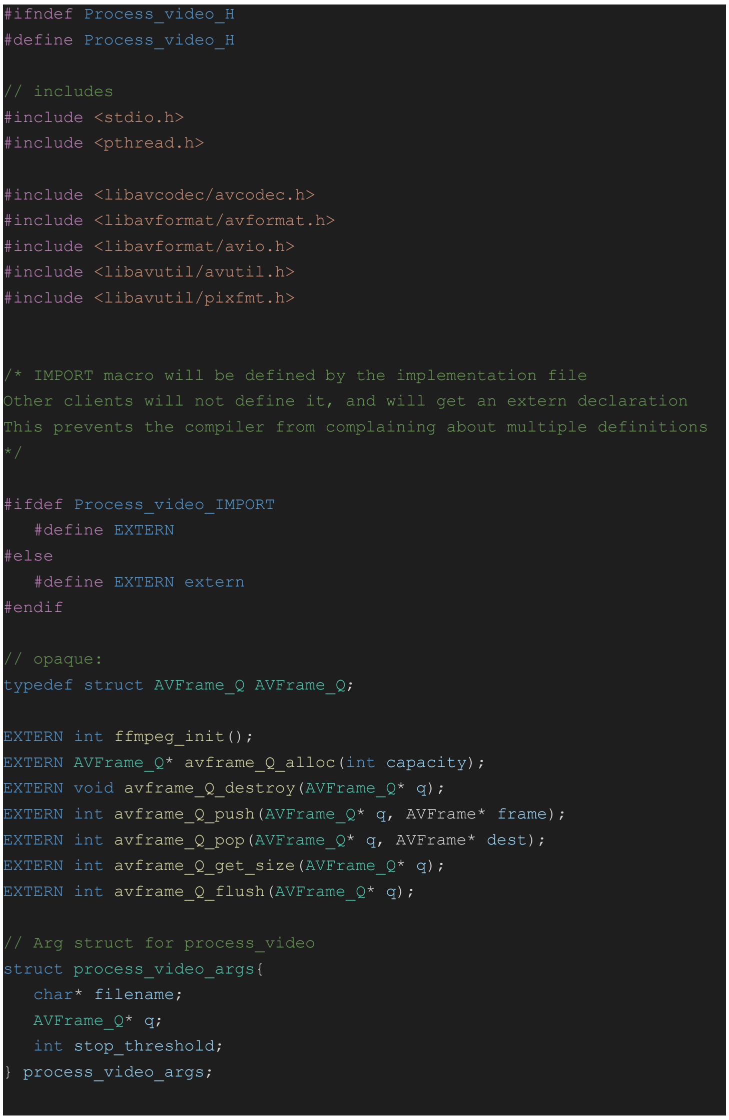
VSCODE setup: Ditt og datt.
Endret permission på opt/DIS folder for å kunne endre filer og adde konfigurasjoner (må til for å la VScode kjøre med custom include paths etc. Alternativet er å kjøre manuelt med terminalinput)

C Makefile

```makefile
# Makefile
CC = gcc
# Compiler flags: all warnings + debugger meta-data
CFLAGS = -Wall -g -m64 -Wall -D_REENTRANT
INCLUDES=-Iinclude -I/opt/DIS/include -I/opt/DIS/src/include -I/opt/DIS/include/dis

#OS Specific?
BITS=$(shell sh -c 'getconf LONG_BIT || echo NA')

ifeq ($(BITS),NA)
LIB_DIRECTORY=lib
endif

ifeq ($(BITS),32)
LIB_DIRECTORY=lib
endif

ifeq ($(BITS),64)
LIB_DIRECTORY=lib64
endif
# External libraries:
LIBS = -Wl,-L/opt/DIS/$(LIB_DIRECTORY),-rpath,/opt/DIS/$(LIB_DIRECTORY),-lsisci -lSDL2 -lavcodec -lavformat -lavutil -lswscale -lpthread

# Pre-defined macros for conditional compilation
DEFS = -DDEBUG_FLAG -DEXPERIMENTAL=0

# The final executable program file, i.e. name of our program
BIN = server
BIN2 = client

# Object files from which $BIN depends
SERVER_OBJS = process_video.o videoplayer.o
CLIENT_OBJS = videoplayer.o

$(BIN): $(SERVER_OBJS) $(BIN).c
	$(CC) $(CFLAGS) $(DEFS) $(INCLUDES) $(SERVER_OBJS) $(BIN).c $(LIBS) -o $(BIN)

$(BIN2): $(CLIENT_OBJS) $(BIN2).c
	$(CC) $(CFLAGS) $(DEFS) $(INCLUDES) $(CLIENT_OBJS) $(BIN2).c $(LIBS) -o $(BIN2)

%.o: %.c %.h
	$(CC) -c $(CFLAGS) $(DEFS) $(INCLUDES) $< -o $@
```

```
clean:
	rm -f *~ *.o $(BIN)

depend:
	makedepend -Y -- $(CFLAGS) $(DEFS) -- *.c
```

D Video processor module interface



```c
#ifndef Process_video_H
#define Process_video_H

// includes
#include <stdio.h>
#include <pthread.h>

#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavformat/avio.h>
#include <libavutil/avutil.h>
#include <libavutil/pixfmt.h>


/* IMPORT macro will be defined by the implementation file
Other clients will not define it, and will get an extern declaration
This prevents the compiler from complaining about multiple definitions
*/

#ifdef Process_video_IMPORT
   #define EXTERN
#else
   #define EXTERN extern
#endif

// opaque:
typedef struct AVFrame_Q AVFrame_Q;

EXTERN int ffmpeg_init();
EXTERN AVFrame_Q* avframe_Q_alloc(int capacity);
EXTERN void avframe_Q_destroy(AVFrame_Q* q);
EXTERN int avframe_Q_push(AVFrame_Q* q, AVFrame* frame);
EXTERN int avframe_Q_pop(AVFrame_Q* q, AVFrame* dest);
EXTERN int avframe_Q_get_size(AVFrame_Q* q);
EXTERN int avframe_Q_flush(AVFrame_Q* q);

// Arg struct for process_video
struct process_video_args{
   char* filename;
   AVFrame_Q* q;
   int stop_threshold;
} process_video_args;
```

```
// Function prototypes
EXTERN void* process_video(void* arg);


#undef Process_video_IMPORT
#undef EXTERN
#endif
```

E Video player module interface

```c
#ifndef Videoplayer_H
#define Videoplayer_H

// includes
#include <stdio.h>
#include <libavcodec/avcodec.h>
#include <libavutil/imgutils.h>
#include <libavutil/mem.h>
#include <libavutil/pixfmt.h>


// include SDL
#include <SDL2/SDL.h>
#include <SDL2/SDL_thread.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL_timer.h>
#include <SDL2/SDL_video.h>
#include <SDL2/SDL_render.h>
#include <SDL2/SDL_pixels.h>

#define FRAME_RATE 30
#define FRAME_DELAY 1000/FRAME_RATE
// IMPORT macro will be defined by the implementation file
// Other clients will not define it, and will get an extern declaration
// This prevents the compiler from complaining about multiple
definitions

#ifdef Videoplayer_IMPORT
   #define EXTERN
#else
   #define EXTERN extern
#endif

//Struct for frame data.
//Writing frame data without context over PCIe makes reconstruction
impossible
//if we dont know the exact format.
//We therefore need a metadata struct to send with the frame data.
typedef struct {
   int width;
   int height;
   enum AVPixelFormat pix_fmt;
   int buffer_size;
   int linesize[AV_NUM_DATA_POINTERS];
   int align;
} FrameData;

// Reconstruction function for after PCIe transfer
int update_avframe_from_framedata(FrameData *framedata, AVFrame *frame,
void *data_addr);
int update_framedata_from_avframe(AVFrame *frame, FrameData
*framedata);

typedef struct VideoPlayer {
   SDL_Window *window;
   SDL_Renderer *renderer;
} VideoPlayer;

// opaque:
EXTERN int create_videoplayer(VideoPlayer* vp);
EXTERN void update_videoplayer(VideoPlayer* vp, AVFrame *frame,
uint32_t delay_time);
EXTERN void destroy_videoplayer(VideoPlayer* vp);


// Function prototypes


#undef Videoplayer_IMPORT
#undef EXTERN
#endif
```