Bachelors Thesis

Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Anatol Constantin Fürst

28th January 2024

Technische Universität Dresden
Faculty of Computer Science
Institute of Systems Architecture
Chair of Operating Systems

Academic Supervisors:
Prof. Dr.-Ing. Horst Schirmeier
Prof. Dr.-Ing. habil. Dirk Habich
M.Sc. André Berthold

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Aufgabenstellung für die Anfertigung einer Bachelor-Arbeit

Studiengang:	Bachelor
Studienrichtung:	Informatik (2009)
Name:	Constantin Fürst
Matrikelnummer:	4929314
Titel:	Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Developments in main memory technologies like Non-Volatile RAM (NVRAM), High Bandwidth Memory (HBM), NUMA, or Remote Memory, lead to heterogeneous memory systems that, instead of providing one monolithic main memory, deploy multiple memory devices with different non-functional memory properties. To reach optimal performance on such systems, it becomes increasingly important to move data, ahead of time, to the memory device with non-functional properties tailored for the intended workload, making data movement operations increasingly important for data intensive applications. Unfortunately, while copying, the CPU is mostly busy with waiting for the main memory, and cannot work on other computations. To tackle this problem Intel implements the Intel Data Streaming Accelerator (Intel DSA), an engine to explicitly offload data movement operations from the CPU, in their newly released Intel Xeon CPU Max processors.

The goal of this bachelor thesis is to analyze and characterize the architecture of the Intel DSA and the vendor-provided APIs. The student should benchmark the performance of Intel DSA and compare it to the CPU's performance, concentrating on data transfers between DDR5-DRAM and HBM and between different NUMA nodes. Additionally, the student should find out in what way and to what extent parallel processes copying data interfere with each other. Analyzing the performance information, the thesis should outline a gainful utilization of the Intel DSA and demonstrate its potential by extending the Query-driven Prefetching concept, which aims to speed up database query execution in heterogeneous memory systems.

Gutachter:	Prof. Dr.-Ing. Dirk Habich
Betreuer:	André Berthold, M.Sc.
Ausgehändigt am:	4. Dezember 2023
Einzureichen am:	19. Februar 2024

Prof. Dr.-Ing. Horst Schirmeier
Betreuender Hochschullehrer

Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Dresden, 28th January 2024

Anatol Constantin Fürst

Abstract

...abstract ...

write the abstract

Contents

List of Figures **XI**

1 Introduction **1**

2 Technical Background **3**
2.1 High Bandwidth Memory . . . 3
2.2 Query-driven Prefetching . . . 3
2.3 Intel Data Streaming Accelerator . . . 3
2.4 Programming Interface for Intel Data Streaming Accelerator . . . 7
2.5 System Setup and Configuration . . . 8

3 Performance Microbenchmarks **11**
3.1 Benchmarking Methodology . . . 11
3.2 Benchmarks . . . 13
3.3 Analysis . . . 18

4 Design **21**
4.1 Detailed Task Description . . . 21
4.2 Cache Design . . . 21
4.3 Accelerator Usage . . . 24

5 Implementation **25**
5.1 Locking and Usage of Atomics . . . 25
5.2 Accelerator Usage . . . 30
5.3 Application to Query-driven Prefetching . . . 30

6 Evaluation **31**
6.1 Expectations . . . 31
6.2 Observation and Discussion . . . 31

7 Conclusion And Outlook **33**
7.1 Conclusions . . . 33
7.2 Future Work . . . 33

Glossary **35**

Bibliography **37**

List of Figures

2.1 Intel Data Streaming Accelerator Internal Architecture [5, Fig. 1 (a)]. Shows the components that the chip is made up of, how they are connected and which outside components the DSA communicates with. 4
2.2 Intel Data Streaming Accelerator Software View [5, Fig. 1 (b)]. Illustrating the software stack and internal interactions from user applications, through the driver to the portal for work submission. 6
2.3 Memcpy Implementation Pseudocode. Performs copy operation of a block of memory from source to destination. The DSA executing this copy can be selected with the parameter `node`, and the template parameter `path` elects whether to run on hardware (Intel DSA) or software (CPU). . . . 7

3.1 Xeon Max Layout [12, Fig. 14] for a 2-Socket System when configured with HBM-Flat. Showing separate Node IDs for manual HBM access and for Cores and DDR-SDRAM. 11
3.2 Benchmark Procedure Pseudocode. Timing marked with yellow background. Showing data allocation and the benchmarking loop for a single thread. 12
3.3 Throughput for different Submission Methods and Sizes. Performing a copy with source and destination being node 0, executed by the DSA on node 0. Observable is the submission cost which affects small transfer sizes differently, as there the completion time is lower. 13
3.4 Throughput for different Thread Counts and Sizes. Multiple threads submit to the same Shared Work Queue. Performing a copy with source and destination being node 0, executed by the DSA on node 0. 14
3.5 Throughput for brute force copy from DDR-SDRAM to HBM. Using all available DSA. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. Shows peak throughput achievable with DSA. . 16
3.6 Throughput for smart copy from DDR-SDRAM to HBM. Using four on-socket DSA for intra-socket operation and the DSA on source and destination node for inter-socket. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. Shows conservative performance. 16
3.7 Throughput from DDR-SDRAM to HBM on CPU. Using the exact same code as smart copy, therefore spawning one thread on Node 0, 1, 2 and 3. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. This shows the low performance of software path, when not adapting the code. 17

3.8 Throughput from Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM) to High Bandwidth Memory (HBM) on CPU. Using 12 Threads spawned on NUMA-Node (Node) 0 for the task. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. 18

4.1 Public Interface of `CacheData` and `Cache` Classes. Colour coding for thread safety. Grey denotes impossibility for threaded access. Green indicates full safety guarantees only relying on atomics to achieve this. Yellow may use locking but is still safe for use. Red must be called from a single threaded context. 22

5.1 Sequence for Blocking Scenario. Observable in first draft implementation. Scenario where T_1 performed first access to a datum followed T_2 and T_3. Then T_1 holds the handlers exclusively, leading to the other threads having to wait for T_1 to perform the work submission and waiting before they can access the datum through the cache. 27
5.2 `CacheData::WaitOnCompletion` Pseudocode. Final rendition of the implementation for a fair wait function. 28

6.1 Illustration of the benchmarked simple query in (a) and the corresponding pipeline in (b). Taken from [25, Fig. 1]. 31

1 Introduction

write this chapter

2 Technical Background

write introductory paragraph

2.1 High Bandwidth Memory

High Bandwidth Memory is a novel memory technology that promises an increase in peak bandwidth. It consists of stacked DDR-SDRAM dies [1, p. 1] and is gradually being integrated into server processors, with the Intel® Xeon® Max Series [2] being one recent example. HBM on these systems can be configured in different memory modes, most notably, HBM Flat Mode and HBM Cache Mode [2]. The former gives applications direct control, requiring code changes, while the latter utilizes the HBM as a cache for the system's DDR-SDRAM-based main memory [2].

2.2 Query-driven Prefetching

write this section

2.3 Intel Data Streaming Accelerator

> Intel DSA is a high-performance data copy and transformation accelerator that will be integrated in future Intel® processors, targeted for optimizing streaming data movement and transformation operations common with applications for high-performance storage, networking, persistent memory, and various data processing applications. [3, Ch. 1]

Introduced with the 4^{th} generation of Intel Xeon Scalable Processors, the DSA aims to relieve the CPU from 'common storage functions and operations such as data integrity checks and deduplication' [4, p. 4]. To fully utilize the hardware, a thorough understanding of its workings is essential. Therefore, we present an overview of the architecture, software, and the interaction of these two components, delving into the architectural details of the DSA itself. All statements are based on Chapter 3 of the Architecture Specification by Intel.

2.3.1 Hardware Architecture

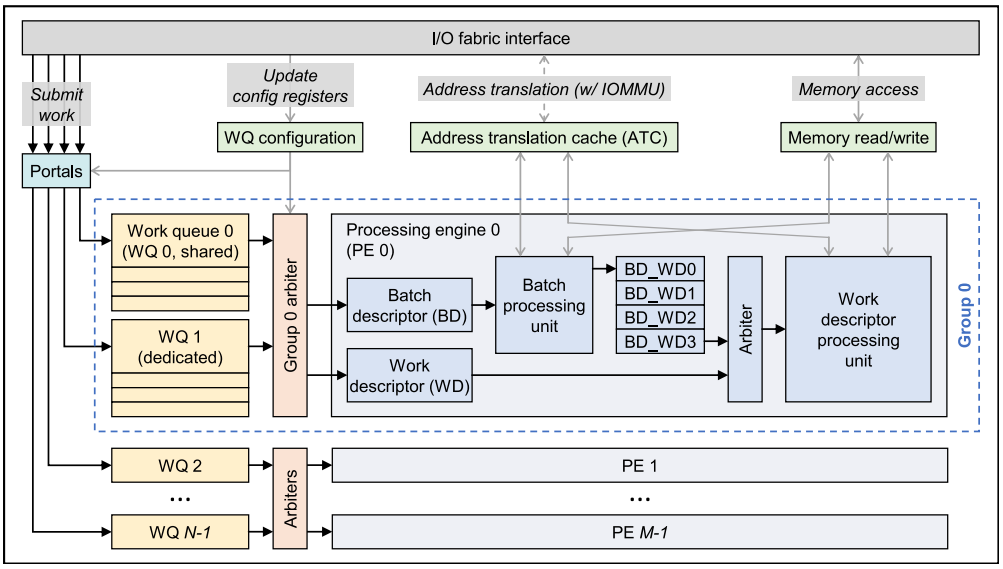

Figure 2.1: Intel Data Streaming Accelerator Internal Architecture [5, Fig. 1 (a)]. Shows the components that the chip is made up of, how they are connected and which outside components the DSA communicates with.

The DSA chip is directly integrated into the processor and attaches via the I/O fabric interface, serving as the conduit for all communication. Through this interface, the DSA is accessible as a PCIe device. Consequently, configuration utilizes memory-mapped registers set in the devices Base Address Register (BAR). Through these registers, the devices' layout is defined and memory pages for work submission set. In a system with multiple processing nodes, there may also be one DSA per node, resulting in up to four DSA devices per socket in 4^{th} generation Intel Xeon Processors [6, Sec. 3.1.1]. To accommodate various use cases, the layout of the DSA is software-defined. The structure comprises three components, which we will describe in detail. We also briefly explain how the DSA resolves virtual addresses and signals operation completion. At last, we will detail operation execution ordering.

2.3.1.1 Architectural Components

Component I Work Queue: Work Queue (WQ)s provide the means to submit tasks to the device and will be described in more detail shortly. They are marked yellow in Figure 2.1. A WQ is accessible through so-called portals, light blue in Figure 2.1, which are mapped memory regions. Submission of work is done by writing a descriptor to one of these. A descriptor is 64 bytes in size and may contain one specific task (task descriptor) or the location of a task array in memory (batch descriptor). Through these portals, the submitted descriptor reaches a queue. There are two possible queue types with different submission methods and use cases. The Shared Work Queue (SWQ) is intended to provide synchronized access to multiple processes and each group may only

have one attached. A PCIe Deferrable Memory Write Request (DMR), which guarantees implicit synchronization, is generated via x86 Instruction ENQCMD and communicates with the device before writing [3, Sec. 3.3.1]. This may result in higher submission cost, compared to the Dedicated Work Queue (DWQ) to which a descriptor is submitted via x86 Instruction MOVDIR64B [3, Sec. 3.3.2].

Component II Engine: An Engine is the processing-block that connects to memory and performs the described task. To handle the different descriptors, each Engine has two internal execution paths. One for a task and the other for a batch descriptor. Processing a task descriptor is straightforward, as all information required to complete the operation are contained within . For a batch, the DSA reads the batch descriptor, then fetches all task descriptors from memory and processes them [3, Sec. 3.8]. An Engine can coordinate with the operating system in case it encounters a page fault, waiting on its resolution, if configured to do so, while otherwise, an error will be generated in this scenario [3, Sec. 2.2, Block on Fault].

cite this

Component III Groups: Groups tie Engines and Work Queues together, indicated by the dotted blue line around the components of Group 0 in Figure 2.1. This means, that tasks from one WQ may be processed from multiple Engines and vice-versa, depending on the configuration. This flexibility is achieved through the Group Arbiter, represented by the orange block in Figure 2.1, which connects the two components according to the user-defined configuration.

2.3.1.2 Virtual Address Resolution

An important aspect of modern computer systems is the separation of address spaces through virtual memory. Therefore, the DSA must handle address translation because a process submitting a task will not know the physical location in memory, causing the descriptor to contain virtual addresses. To resolve these to physical addresses, the Engine communicates with the Input/Output Memory Management Unit (IOMMU) to perform this operation, as visible in the outward connections at the top of Figure 2.1. Knowledge about the submitting processes is required for this resolution. Therefore, each task descriptor has a field for the Process Address Space ID (PASID) which is filled by the ENQCMD instruction for a SWQ [3, Sec. 3.3.1] or set statically after a process is attached to a DWQ [3, Sec. 3.3.2].

2.3.1.3 Completion Signalling

The status of an operation on the DSA is available in the form of a record, which is written to a memory location specified in the task descriptor. Applications can check for a change in value in this record to determine completion. Additionally, completion may be signalled by an interrupt. To facilitate this, the DSA ‘provides two types of interrupt message storage: (1) an MSI-X table, enumerated through the MSI-X capability; and (2) a device-specific Interrupt Message Storage (IMS) table’ [3, Sec. 3.7].

2.3.1.4 Ordering Guarantees

Ordering of operations is only guaranteed for a configuration with one WQ and one Engine in a Group when exclusively submitting batch or task descriptors but no mixture. Even in such cases, only write-ordering is guaranteed, implying that 'reads by a subsequent descriptor can pass writes from a previous descriptor'. Challenges arise, when an operation fails, as the DSA will continue to process the following descriptors from the queue. Consequently, caution is necessary in read-after-write scenarios. This can be addressed by either waiting for successful completion before submitting the dependent descriptor, inserting a drain descriptor for tasks, or setting the fence flag for a batch. The latter two methods inform the processing engine that all writes must be committed, and in case of the fence in a batch, to abort on previous error. [3, Sec. 3.9]

2.3.2 Software View

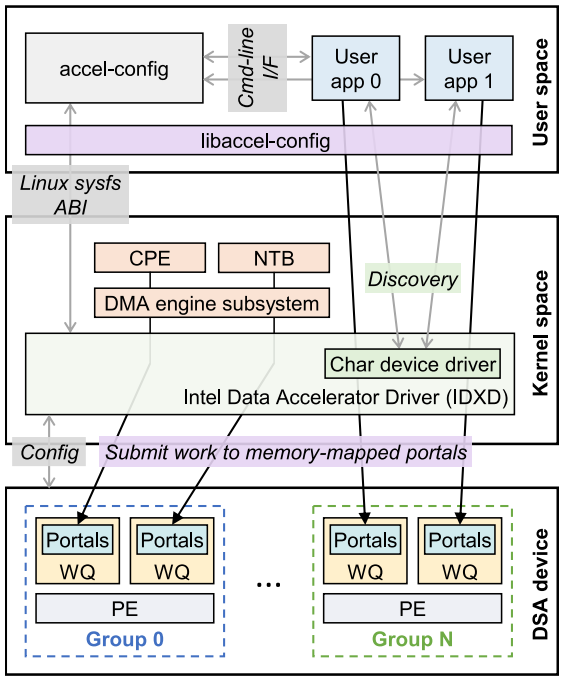

Figure 2.2: Intel Data Streaming Accelerator Software View [5, Fig. 1 (b)]. Illustrating the software stack and internal interactions from user applications, through the driver to the portal for work submission.

Since Linux Kernel 5.10, there exists a driver for the DSA which has no counterpart in the Windows OS-Family [7, Sec. Installation] and other operating systems. Therefore, accessing the DSA is only possible under Linux. To interact with the driver and perform configuration operations, Intel's accel-config [8] user-space toolset can be utilized. This application provides a command-line interface and can read configuration files to set up the device. The interaction is depicted in the upper block titled 'User space' in Figure

2.2. It interacts with the kernel driver, visible in light green and labelled 'IDXD' in Figure 2.2. After successful configuration, each WQ is exposed as a character device through `mmap` of the associated portal [5, Sec. 3.3].

With the appropriate file permissions, a process could submit work to the DSA using either the MOVDIR64B or ENQCMD instructions, providing the descriptors by manual configuration. However, this process can be cumbersome, which is why Intel Data Mover Library (Intel DML) exists.

With some limitations, like lacking support for DWQ submission, this library presents an interface that takes care of creation and submission of descriptors, and error handling and reporting. Thanks to the high-level-view the code may choose a different execution path at runtime which allows the memory operations to either be executed in hardware or software. The former on an accelerator or the latter using equivalent instructions provided by the library. This makes code using this library automatically compatible with systems that do not provide hardware support. [7, Sec. Introduction]

2.4 Programming Interface for Intel Data Streaming Accelerator

As mentioned in Subsection 2.3.2, Intel DML offers a high level interface for interacting with the hardware accelerator, specifically Intel DSA. Opting for the C++ interface, we will now demonstrate its usage by example of a simple memcopy implementation for the DSA.

```
template <path>
bool DsaMemcpy(char* dst, const char* src, size_t size, int node)

numa_run_on_node(node)

src_view ← dml::make_view(src, size)
dst_view ← dml::make_view(dst, size)

handler ← dml::submit<path>(dml::mem_copy.block_on_fault(), src_view, dst_view)

result ← handler.get()

return result.status == dml::status_code::ok
```

Figure 2.3: Memcpy Implementation Pseudocode. Performs copy operation of a block of memory from source to destination. The DSA executing this copy can be selected with the parameter `node`, and the template parameter `path` elects whether to run on hardware (Intel DSA) or software (CPU).

In the function header of Figure 2.3 two differences from standard memcpy are notable. Firstly, there is the template parameter named `path`, and secondly, an additional parameter `int node`. Both will be discussed in the following paragraphs.

The `path` parameter allows the selection of the executing device, which can be either the CPU or DSA. The options include `dml::software` (CPU), `dml::hardware` (DSA), and `dml::automatic`, where the latter dynamically selects the device at runtime, favoring DSA over CPU execution [7, Sec. Quick Start].

Choosing the engine which carries out the copy might be advantageous for performance, as we can see in Subsection 3.2.3. With the engine directly tied to the processing node, as observed in Subsection 2.3.1, the node ID is equivalent to the ID of the DSA.

Intel DML operates on data views, which we create from the given pointers to source and destination and size. This is done using `dml::make_view(uint8_t* ptr, size_t size)`, visible in Figure 2.3, where these views are labelled `src_view` and `dst_view`. [7, Sec. High-level C++ API, Make view]

In Figure 2.3, we submit a single descriptor using the asynchronous operation from Intel DML. This uses the function `dml::submit<path>`, which takes an operation type and parameters specific to the selected type and returns a handler to the submitted task. For the copy operation, we pass the two views created previously. The provided handler can later be queried for the completion of the operation. After submission, we poll for the task completion with `handler.get()` and check whether the operation completed successfully.

A noteworthy addition to the submission-call is the use of `.block_on_fault()`, enabling the DSA to manage a page fault by coordinating with the operating system. It's essential to highlight that this functionality only operates if the device is configured to accept this flag. [7, Sec. High-level C++ API, How to Use the Library] [7, Sec. High-level C++ API, Page Fault handling]

2.5 System Setup and Configuration

In this section we provide a step-by-step guide to replicate the configuration being used for benchmarks and testing purposes in the following chapters. While Intel's guide on DSA usage was a useful resource, we also consulted articles for setup on Lenovo ThinkSystem Servers for crucial information not present in the former. It is important to note that instructions for configuring the HBM access mode, as mentioned in Section 2.1, may vary from system to system and can require extra steps not covered in the list below.

1. Set 'Memory Hierarchy' to Flat [9, Sec. Configuring HBM, Configuring Flat Mode], 'VT-d' to Enabled in BIOS [6, Sec. 2.1] and, if available, 'Limit CPU PA to 46 bits' to Disabled in BIOS [10, p. 5]

2. Use a kernel with IDXD driver support, available from Linux 5.10 or later [7, Sec. Installation] and append the following to the kernel boot parameters in grub config: `intel_iommu=on,sm_on` [10, p. 5]

3. Evaluate correct detection of DSA devices using `dmesg | grep idxd` which should list as many devices as NUMA nodes on the system [10, p. 5]

4. Install `accel-config` from repository [8] or system package manager and inspect the detection of DSA devices through the driver using `accel-config list -i` [10, p. 6]

5. Create DSA configuration file for which we provide an example under `benchmarks/configuration-files/8n1d1e1w.conf` in the accompanying repository [11] that is used for most benchmarks available. Then apply the configuration using `accel-config load-config -c [filename] -e` [6, Fig. 3-9]

6. Inspect the now configured DSA devices using `accel-config list` [10, p. 7], output should match the desired configuration set in the file used

3 Performance Microbenchmarks

In this chapter, we measure the performance of the DSA, with the goal to determine an effective utilization strategy to apply the DSA to Query-driven Prefetching (QdP). In Section 3.1 we lay out our benchmarking methodology, then perform benchmarks in 3.2 and finally summarize our findings in 3.3.

The performance of DSA has been evaluated in great detail by Reese Kuper et al. in [5]. Therefore, we will perform only a limited amount of benchmarks with the purpose of verifying the figures from [5] and analysing best practices and restrictions for applying DSA to QdP.

3.1 Benchmarking Methodology

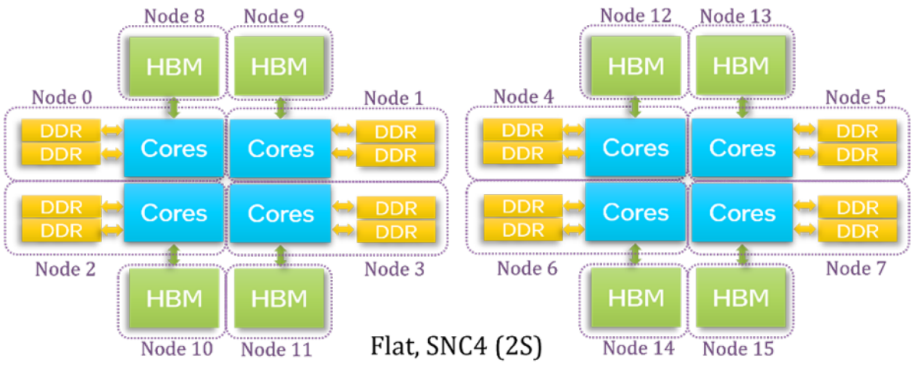

Figure 3.1: Xeon Max Layout [12, Fig. 14] for a 2-Socket System when configured with HBM-Flat. Showing separate Node IDs for manual HBM access and for Cores and DDR-SDRAM.

The benchmarks were conducted on a dual-socket server equipped with two Intel Xeon Max 9468 CPUs, each with 4 nodes that have access to 16 GiB of HBM and 12 cores. This results in a total of 96 cores and 128 GiB of HBM. The layout of the system is visualized in Figure 3.1. For configuring it, we follow Section 2.5.

As Intel DML does not have support for DWQs, we run benchmarks exclusively with access through SWQs. The application written for the benchmarks can be obtained in source form under the directory `benchmarks` in the thesis repository [11].

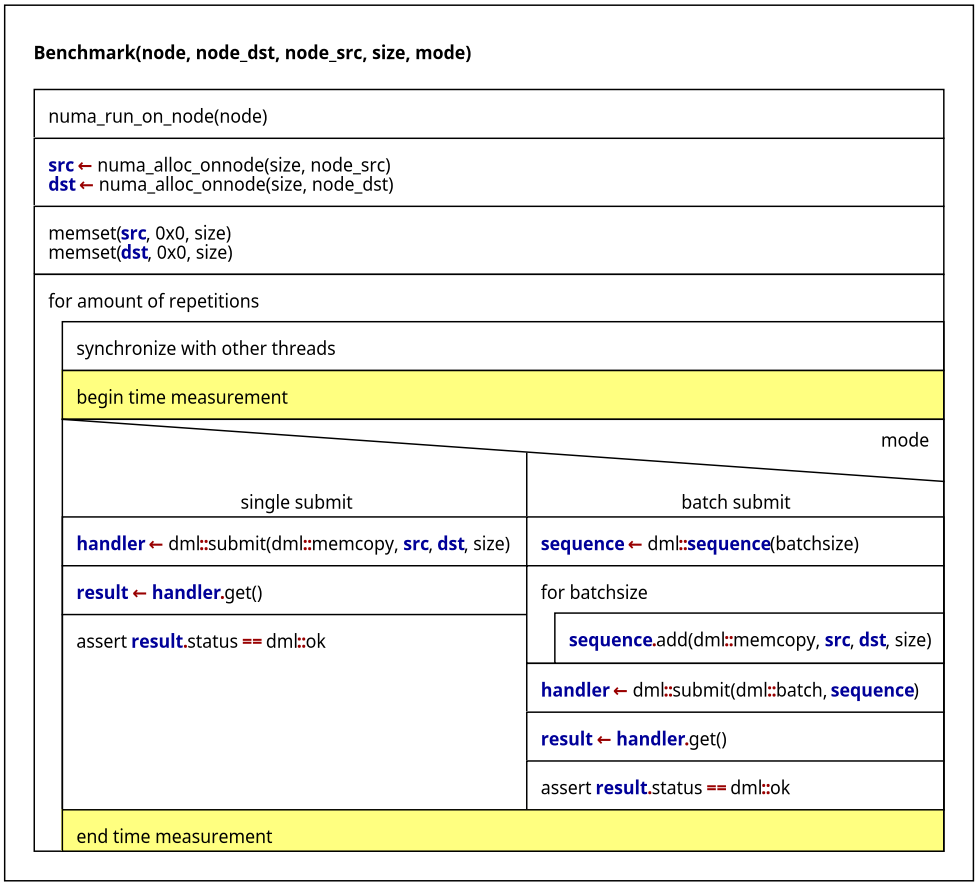

Figure 3.2: Benchmark Procedure Pseudocode. Timing marked with yellow background. Showing data allocation and the benchmarking loop for a single thread.

The benchmark performs node setup as described in Section 2.4 and allocates source and destination memory on the nodes passed in as parameters. To avoid page faults affecting the results, the entire memory regions are written to before the timed part of the benchmark starts. To get accurate results, the benchmark is repeated at least 100 times. Each iteration is timed from beginning to end, marked by yellow in Figure 3.2. The launch is synchronized by use of a barrier for each iteration. The behaviour then differs depending on the submission method selected which can be a single submission or a batch of given size. This can be seen in Figure 3.2 at the switch statement for 'mode'. Single submission follows the example given in Section 2.4, and we therefore do not go into detail explaining it here. Batch submission works unlike the former. A sequence with specified size is created which tasks are then added to. This sequence is submitted to the engine similar to the submission of a single descriptor.

3.2 Benchmarks

In this section we will In this section, we will present three benchmarks, each accompanied by setup information and a preview. We will then provide plots displaying the results, followed by a detailed analysis. We will formulate expectations and compare them with the observations from our measurements.

3.2.1 Submission Method

With each submission, descriptors must be prepared and sent to the underlying hardware. This process is anticipated to incur a cost, impacting throughput sizes and submission methods differently. We submit different sizes and compare batching with single submissions, determining which combination of submission method and size is most effective.

We anticipate that single submissions will consistently yield poorer performance, particularly with a pronounced effect on smaller transfer sizes. This expectation arises from the fact that the overhead of a single submission with the SWQ is incurred for every iteration, whereas the batch experiences this overhead only once for multiple copies.

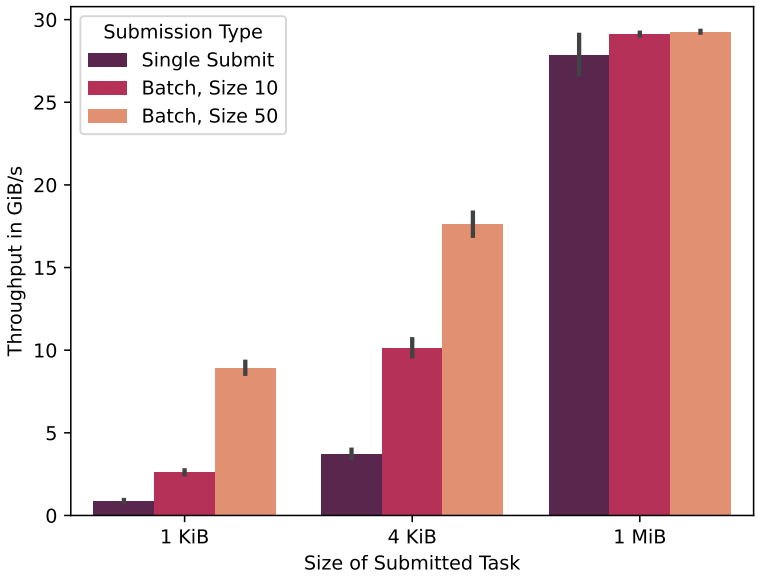

Figure 3.3: Throughput for different Submission Methods and Sizes. Performing a copy with source and destination being node 0, executed by the DSA on node 0. Observable is the submission cost which affects small transfer sizes differently, as there the completion time is lower.

In Figure 3.3 we conclude that with transfers of 1 MiB and upwards, the submission method makes no noticeable difference. For smaller transfers the performance varies greatly, with batch operations leading in throughput. This finding is aligned with the

observation that 'SWQ observes lower throughput between 1-8 KB [transfer size]' [5, p. 6 and 7] for normal submission method.

Another limitation may be observed in this result, namely the inherent throughput limit per DSA chip of close to 30 GiB/s. This is apparently caused by I/O fabric limitations [5, p. 5]. We therefore conclude, that the use of multiple DSA is required to fully utilize the available bandwidth of HBM which theoretically lies at 256 GB/s [1, Table I].

3.2.2 Multithreaded Submission

As we might encounter access to one DSA from multiple threads through the associated Shared Work Queue, understanding the impact of this type of access is crucial. We benchmark multithreaded submission for one, two, and twelve threads - the latter representing the core count of one processing sub-node on the test system. Each configuration is assigned the same 120 copy tasks split across the available threads, all submitting to one DSA. We perform this benchmark with sizes of 1 MiB and 1 GiB to examine, if the behaviour changes with submission size. For smaller sizes, the completion time may be faster than submission time, leading to potentially different effects of threading due to the fact that multiple threads work to fill the queue, preventing task starvation. We may also experience lower-than-peak throughput with rising thread count, caused by the synchronization inherent with SWQ.

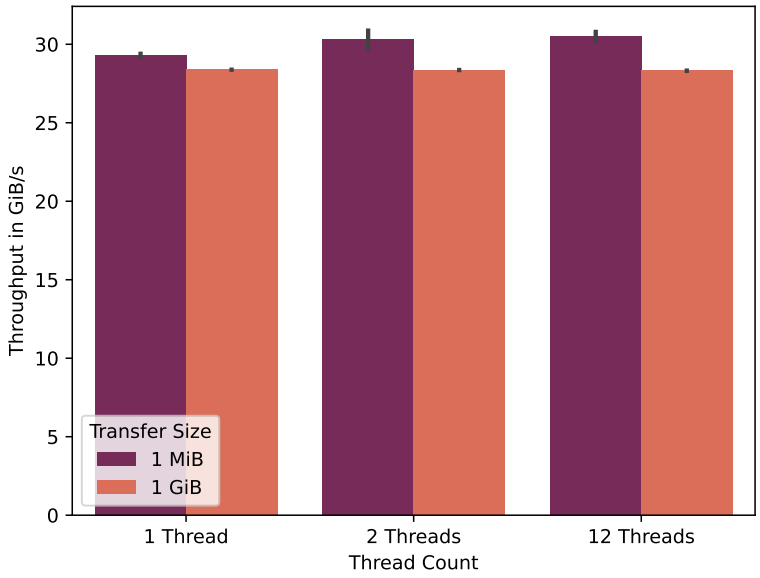

Figure 3.4: Throughput for different Thread Counts and Sizes. Multiple threads submit to the same Shared Work Queue. Performing a copy with source and destination being node 0, executed by the DSA on node 0.

In Figure 3.4, we note that threading has no discernible negative impact. The synchronization appears to affect single-threaded access in the same manner as it does

for multiple threads. Interestingly, for the smaller size of 1 MiB, our assumption proved accurate, and performance actually increased with the addition of threads, which we attribute to enhanced queue usage. However, we were unable to identify an explanation for the speed difference between sizes. This finding contradicts the rationale that higher transfer sizes would result in less impact on submission time and, consequently, higher throughput.

3.2.3 Data Movement from DDR-SDRAM to HBM

Moving data from DDR-SDRAM to HBM is most relevant to the rest of this work, as it is the target application. As we discovered in Section 3.2.1, one DSA has a peak bandwidth limit of 30 GiB/s. We write to HBM with its theoretical peak of 256 GB/s [1, Table I]. Our top speed is therefore limited by the slower main memory. For each node, the test system is configured with two DIMMs of DDR5-4800. We calculate the theoretical throughput as follows: $2\ DIMMs * \frac{4800\ Megatransfers}{Second\ and\ DIMM} * \frac{64-bitwidth}{8bits/byte} = 76800\ MT/s = 75\ GiB/s$. We conclude that to achieve closer-to-peak speeds, a copy task has to be split across multiple DSAs.

how to verify this calculation with a citation? is it even correct because we see close to 100 GiB/s throughput?

Two methods of splitting will be evaluated. The first employs a brute force approach, utilizing all available resources for any transfer direction. The second method's behaviour depends on the data source and destination locations. Given that our system consists of multiple sockets, communication crossing between sockets could introduce latency and bandwidth disadvantages. We posit that for intra-socket transfers, utilizing the DSA from the second socket will have only a marginal effect. For transfers crossing sockets, we assume every DSA performs equally worse, prompting us to use only the ones on the destination and source nodes for them being the physically closest to both memory regions. While this choice may result in lower performance, it uses only one-fourth of the engines in the brute force approach for inter-socket transfers and half for intra-socket transfers. This approach also frees up additional chips for other threads to utilize.

cite this

For this benchmark, we transfer 1 Gibibyte of data from node 0 to the destination node, employing the submission method previously described. For each utilized node, we spawn one pinned thread responsible for submission. We present data for nodes 8, 11, 12, and 15. To understand the selection, refer to Figure 3.1, which illustrates the node IDs of the configured systems and the corresponding storage technology. Node 8 accesses the HBM on node 0, making it the physically closest possible destination. Node 11 is located diagonally on the chip, representing the furthest intra-socket operation benchmarked. Nodes 12 and 15 lie diagonally on the second socket's CPU, making them representative of inter-socket transfer operations.

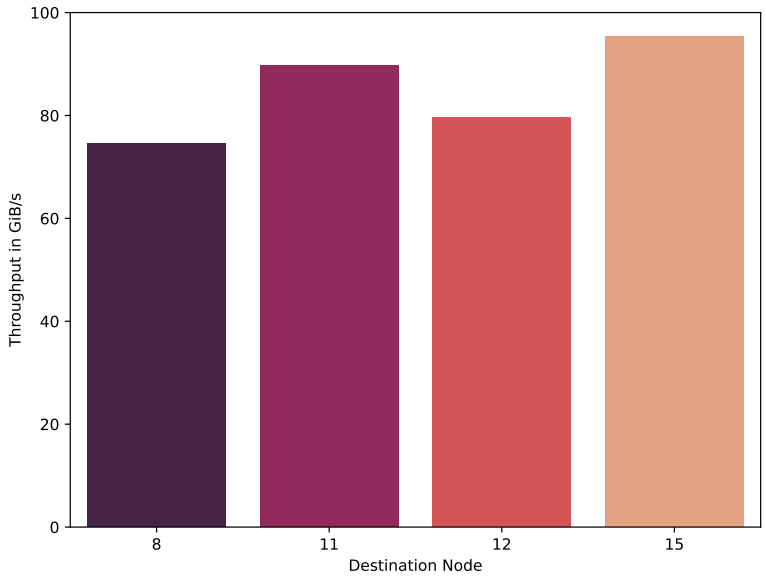

Figure 3.5: Throughput for brute force copy from DDR-SDRAM to HBM. Using all available DSA. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. Shows peak throughput achievable with DSA.

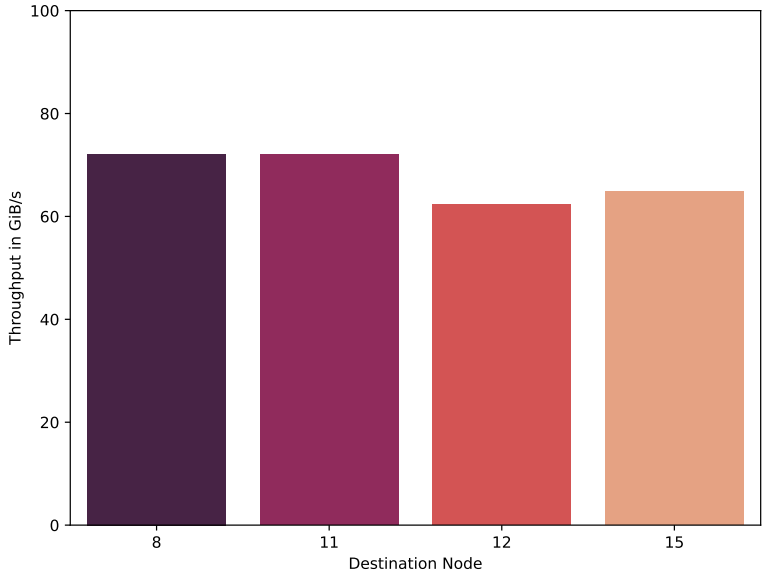

Figure 3.6: Throughput for smart copy from DDR-SDRAM to HBM. Using four on-socket DSA for intra-socket operation and the DSA on source and destination node for inter-socket. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. Shows conservative performance.

From the results of the brute force approach illustrated in Figure 3.5, we observe peak speeds of 96 GiB/s when copying across the socket from Node 0 to Node 15. This contradicts our initial assumption that peak bandwidth would be achieved in the intra-socket scenario and goes against the calculated peak throughput of the memory on our system. Nevertheless, these results align with findings presented in [5, Fig. 10].

find out why, perhaps due to certain factors?

While consuming significantly more resources, the brute force copy depicted in Figure 3.5 surpasses the performance of the smart approach shown in Figure 3.6. We observe an increase in transfer speed by utilizing all available DSA, achieving 2 GiB/s for copying to Node 8, 18 GiB/s for Nodes 11 and 12, and 30 GiB/s for Node 15. The smart approach could accommodate another intra-socket copy on the second socket, we assume, without observing negative impacts. From this, we conclude that the smart copy assignment is worth using, as it provides better scalability.

potential calculation error or other influencing factors?

3.2.4 Data Movement using CPU

For evaluating CPU copy performance two approaches were selected. One requires no code changes, showing the performance of running code developed for DSA on systems where this hardware is not present. For the other approach we modified the code to result in peak performance.

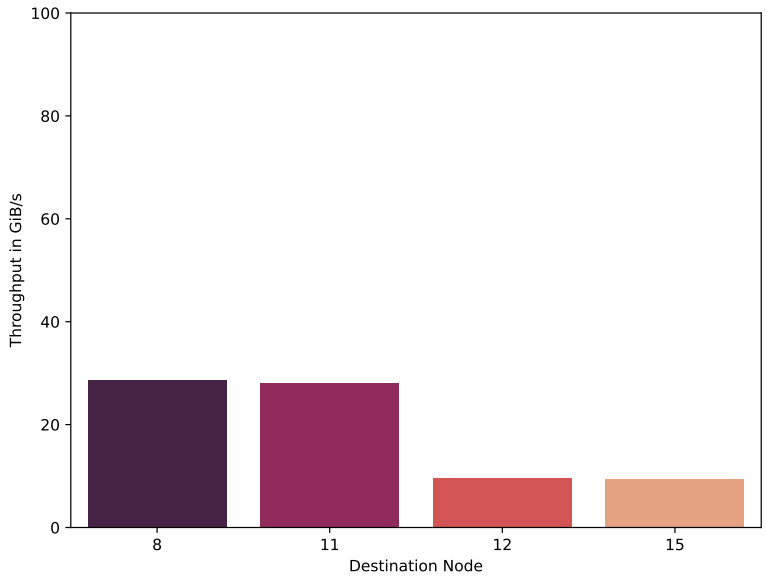

Figure 3.7: Throughput from DDR-SDRAM to HBM on CPU. Using the exact same code as smart copy, therefore spawning one thread on Node 0, 1, 2 and 3. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis. This shows the low performance of software path, when not adapting the code.

For Figure 3.7 we used the smart copy procedure as in 3.2.3, selecting the software path with no other changes to the benchmarking code.

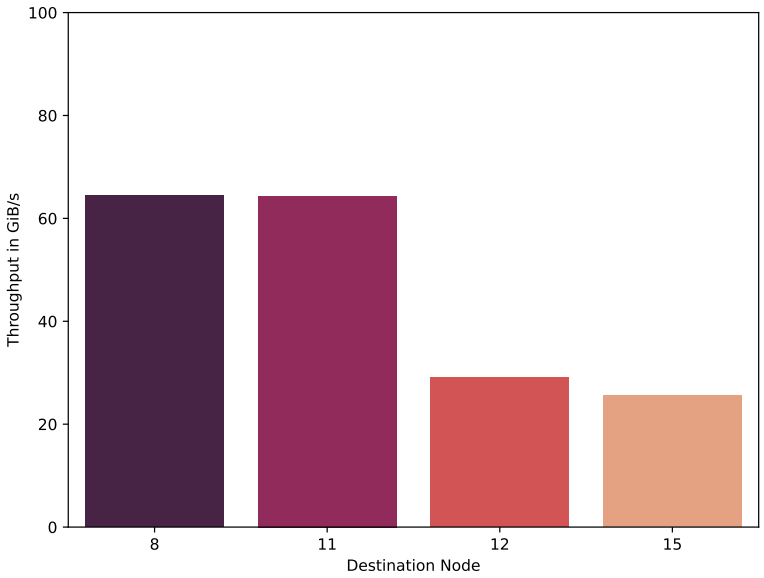

Figure 3.8: Throughput from DDR-SDRAM to HBM on CPU. Using 12 Threads spawned on Node 0 for the task. Copying 1 GiB from Node 0 to the destination Node specified on the x-axis.

The benchmark resulting in Figure 3.8 fully utilizes node 0 of the test system by spawning 12 threads on it. This level of utilization required code modifications beyond selecting a different execution path and would therefore not be achieved without additional programming efforts.

We attribute the low performance of the inter-socket copy operation to overhead of the interconnect between the two sockets. We attribute the slight difference between node 12 and 15 to the on-chip location of these in reference to node 0 which. As Figure 3.1 shows, node 12 is slightly closer to node 0.

Comparing the results in Figure 3.7 with the data from the adapted test displayed in Figure 3.8 we conclude that the software path primarily serves compatibility. The throughput is comparatively low, requiring code changes to increase performance. This leads us to advice against its use.

3.3 Analysis

In this section we summarize the conclusions drawn from the three benchmarks performed in the sections above and outline a utilization guideline. We also compare CPU and DSA for the task of copying data from DDR-SDRAM to HBM.

- From 3.2.1 we conclude that small copies under 1 MiB in size require batching and still do not reach peak performance. Task size should therefore be at or above 1 MiB if possible.

- Subsection 3.2.2 assures that access from multiple threads does not negatively affect the performance when using Shared Work Queue for work submission.
- In 3.2.3, we chose to use the presented smart copy methodology to split copy tasks across multiple DSA chips to achieve low utilization with acceptable performance.

Once again, we refer to Figures 3.5 and 3.8, both representing the maximum throughput achieved with the utilization of either DSA for the former and CPU for the latter. Noticeably, the DSA does not seem to suffer from inter-socket overhead like the CPU. On the contrary, we observe the highest throughput when copying across sockets. In any case, DSA outperforms the CPU for transfer sizes over 1 MiB, demonstrating its potential to increase throughput while simultaneously freeing up CPU cycles.

explanation needed?

4 Design

write introductory paragraph

4.1 Detailed Task Description

write this section

- give slightly more detailed task Description
- perspective of ”what problems have to be solved”
- not ”what is querry driven prefetching”

4.2 Cache Design

The task of prefetching is somewhat aligned with that of a cache. As a cache is more generic and allows use beyond QdP, the decision was made to address the prefetching in QdP by implementing an offloading `Cache`. Henceforth, when referring to the provided implementation, we will use `Cache`.

The interface of `Cache` must provide three basic functions: (1) requesting a memory block to be cached, (2) accessing a cached memory block and (3) synchronizing cache with the source memory. The latter operation comes in to play when the data that is cached may also be modified, necessitating an update either from the source or vice versa. Due various setups and use cases for this cache, the user should also be responsible for choosing cache placement and the copy method. As re-caching is resource intensive, data should remain in the cache for as long as possible. We only flush entries, when lack of free cache memory requires it.

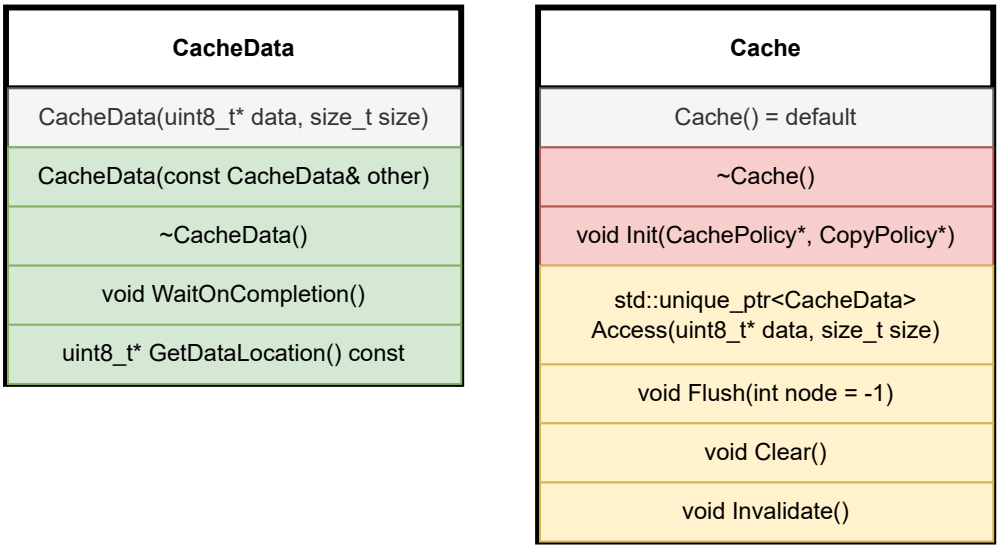

Figure 4.1: Public Interface of `CacheData` and `Cache` Classes. Colour coding for thread safety. Grey denotes impossibility for threaded access. Green indicates full safety guarantees only relying on atomics to achieve this. Yellow may use locking but is still safe for use. Red must be called from a single threaded context.

4.2.1 Interface

To facilitate rapid integration and alleviate developer workload, we opted for a simple interface. Given that this work primarily focuses on caching static data, we only provide cache invalidation and not synchronization. The `Cache::Invalidate` function, given a memory address, will remove all entries for it from the cache. The other two operations, caching and access, are provided in one single function, which we shall henceforth call `Cache::Access`. This function receives a data pointer and size as parameters and takes care of either submitting a caching operation if the pointer received is not yet cached or returning the cache entry if it is. The user retains control over cache placement and the assignment of tasks to accelerators through mechanisms outlined in **??**. This interface is represented on the right block of Figure 4.1 labelled 'Cache' and includes some additional operations beyond the basic requirements.

Given the asynchronous nature of caching operations, users may opt to await their completion. This proves particularly beneficial when parallel threads are actively processing, and the current thread strategically pauses until its data becomes available in faster memory, thereby optimizing access speeds for local computations.

To facilitate this process, the `Cache::Access` method returns an instance of an object referred to as `CacheData`. Figure 4.1 documents the public interface for `CacheData` on the left block labelled as such Invoking `CacheData::GetDataLocation` provides access to a pointer to the location of the cached data. Additionally, the `CacheData::WaitOnCompletion` method is available, designed to return only upon the completion of the caching operation. During this period, the current thread will sleep,

allowing unimpeded progress for other threads. To ensure that only pointers to valid memory regions are returned, this function must be called in order to update the cache pointer. It queries the completion state of the operation, and, on success, updates the cache pointer to the then available memory region.

4.2.2 Cache Entry Reuse

When multiple consumers wish to access the same memory block through the `Cache`, we face a choice between providing each with their own entry or sharing one for all consumers. The first option may lead to high load on the accelerator due to multiple copy operations being submitted and also increases the memory footprint of the cache. The latter option, although more complex, was chosen to address these concerns. To implement this, the existing `CacheData` will be extended in scope to handle multiple consumers. Copies of it can be created, and they must synchronize with each other for `CacheData::WaitOnCompletion` and `CacheData::GetDataLocation`. This is illustrated by the green markings, indicating thread safety guarantees for access, in Figure 4.1.

4.2.3 Cache Entry Lifetime

Allowing multiple references to the same entry introduces concerns regarding memory management. The allocated block should only be freed when all copies of a `CacheData` instance are destroyed, thereby tying the cache entry's lifetime to the longest living copy of the original instance. This ensures that access to the entry is legal during the lifetime of any `CacheData` instance. Therefore, deallocation only occurs when the last copy of a `CacheData` instance is destroyed.

4.2.4 Usage Restrictions

The cache, in the context of this work, primarily handles static data. Therefore, two restrictions are placed on the invalidation operation. This decision results in a drastically simpler cache design, as implementing a fully coherent cache would require developing a thread-safe coherence scheme, which is beyond the scope of our work.

Firstly, overlapping areas in the cache will result in undefined behaviour during the invalidation of any one of them. Only the entries with the equivalent source pointer will be invalidated, while other entries with differing source pointers, which due to their size, still cover the now invalidated region, will remain unaffected. At this point, the cache may or may not continue to contain invalid elements.

Secondly, invalidation is a manual process, requiring the programmer to remember which points of data are currently cached and to invalidate them upon modification. No ordering guarantees are provided in this situation, potentially leading to threads still holding pointers to now-outdated entries and continuing their progress with this data.

Due to its reliance on libnuma for memory allocation, `Cache` is exclusively compatible with systems where this library is available. It is important to note that Windows platforms use their own API for this purpose, which is incompatible with libnuma, rendering the code non-executable on such systems [13].

4.3 Accelerator Usage

Compared with the challenges of ensuring correct entry lifetime and thread safety, the application of DSA for the task of duplicating data is relatively straightforward, thanks in part to Intel DML [7]. Upon a call to `Cache::Access` and determining that the given memory pointer is not present in the cache, work is submitted to the accelerator. However, before proceeding, the desired location for the cache entry must be determined which the user-defined cache placement policy function handles. Once the desired placement is obtained, the copy policy then determines, which nodes should participate in the copy operation. Following Subsection 2.3.1, this is equivalent to selecting the accelerators. The copy tasks are distributed across the participating nodes. As the choice of cache placement and copy policy is user-defined, one possibility will be discussed in Chapter 5.

5 Implementation

In this chapter, we concentrate on specific implementation details, offering an in-depth view of how the design promises outlined in Chapter 4 are realized. Firstly, we delve into the usage of locking and atomics to achieve thread safety. Subsequently, we provide an example of the policy functions alluded to in Section 4.3. Finally, we apply the cache to Query-driven Prefetching.

5.1 Locking and Usage of Atomics

The usage of locking and atomics has proven to be challenging. Their use is performance-critical, and mistakes may lead to deadlock. Consequently, these aspects constitute the most interesting part of the implementation, which is why this chapter will extensively focus on the details of their implementation.

5.1.1 Cache State Lock

To keep track of the current cache state the `Cache` will hold a reference to each currently existing `CacheData` instance. The reason for this is twofold: In Section 4.2 we decided to keep elements in the cache until forced by memory pressure to remove them. Secondly in Subsection 4.2.2 we decided to reuse one cache entry for multiple consumers. The second part requires access to the structure holding this reference to be thread safe when accessing and modifying the cache state in `Cache::Access`, `Cache::Flush` and `Cache::Clear`. The latter two both require unique locking, preventing other calls to `Cache` from making progress while the operation is being processed. For `Cache::Access` the use of locking depends upon the caches state. At first, only a shared lock is acquired for checking whether the given address already resides in cache, allowing other `Cache::Access`-operations to also perform this check. If no entry for the region is present, a unique lock is required as well when adding the newly created entry to cache.

A map-datastructure was chosen to represent the current cache state with the key being the memory address of the entry and as value the `CacheData` instance. As the caching policy is controlled by the user, one datum may be requested for caching in multiple locations. To accommodate this, one map is allocated for each available NUMA-Node of the system. This can be exploited to reduce lock contention by separately locking each Node's state instead of utilizing a global lock. This ensures that `Cache::Access` and the implicit `Cache::Flush` it may cause can not hinder progress of caching operations on other Nodes. Both `Cache::Clear` and a complete `Cache::Flush` as callable by the user will now iteratively perform their respective task per Node state, also allowing other Node to progress.

Even with this optimization, in scenarios where the `Cache` is frequently tasked with flushing and re-caching by multiple threads from the same node, lock contention will negatively impact performance by delaying cache access. Due to passive waiting, this impact might be less noticeable when other threads on the system are able to make progress during the wait.

5.1.2 CacheData Atomicity

The choice made in 4.2.2 necessitates thread-safe shared access to the same resource. The C++ standard library provides `std::shared_ptr<T>`, a reference-counted pointer that is thread-safe for the required operations [14], making it a suitable candidate for this task. Although an implementation using it was explored, it presented its own set of challenges.

As we aim to minimize the time spent in a locked region, only the task is added to the Node's cache state when locked, with the submission taking place outside the locked region. We assume the handlers of Intel DML to be unsafe for access from multiple threads. To achieve the safety for `CacheData::WaitOnCompletion` outlined in 4.2.2, threads need to coordinate which one performs the actual waiting. To avoid queuing multiple copies of the same task, the task must be added before submission. This results in a `CacheData` instance with an invalid cache pointer and no handlers to wait for, presenting an edge case to be considered.

Using `std::shared_ptr<T>` also introduces uncertainty, relying on the implementation to be performant. The standard does not specify whether a lock-free algorithm is to be used, and [15] suggests abysmal performance for some implementations, although the full article is in Korean. No further research was found on this topic.

Therefore, the decision was made to implement atomic reference counting for `CacheData`. This involves providing a custom constructor and destructor wherein a shared (though a standard pointer) atomic integer is either incremented or decremented using atomic fetch sub and add operations [16] to modify the reference count. In the case of a decrease to zero, the destructor was called for the last reference and then performs the actual destruction.

Additionally, the invalid state of `CacheData` is avoided. To achieve this, the waiting algorithm requires the handlers to be contained in an atomic pointer, and the pointer to the cache memory must be atomic too. This enables the use of the atomic wait operation, which is guaranteed by the standard to be more efficient than simply spinning on Compare-And-Swap [17]. Some standard implementations achieve this by yielding after a short spin cycle [18].

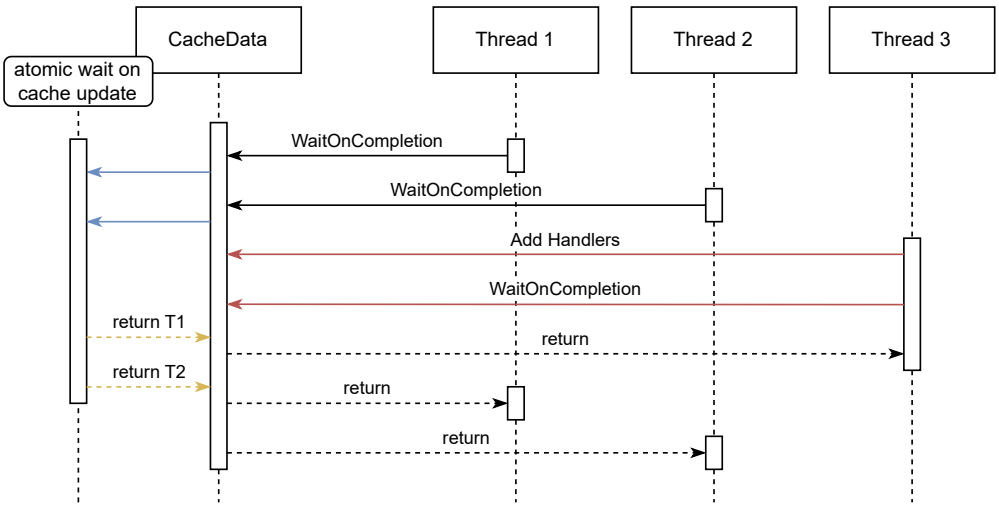

Figure 5.1: Sequence for Blocking Scenario. Observable in first draft implementation. Scenario where T_1 performed first access to a datum followed T_2 and T_3. Then T_1 holds the handlers exclusively, leading to the other threads having to wait for T_1 to perform the work submission and waiting before they can access the datum through the cache.

Designing the wait to work from any thread was complicated. In the first implementation, a thread would check if the handlers are available and if not atomically wait [17] on a value change from nullptr. As the handlers are only available after submission, a situation could arise where only one copy of `CacheData` is capable of actually waiting on them. Lets assume that three threads T_1, T_2 and T_3 wish to access the same resource. T_1 now is the first to call `CacheData::Access` and therefore adds it to the cache state and will perform the work submission. Before T_1 may submit the work, it is interrupted and T_2 and T_3 obtain access to the incomplete `CacheData` on which they wait, causing them to see a nullptr for the handlers but invalid cache pointer, leading to atomic wait on the cache pointer (marked blue lines in Figure 5.1). Now T_1 submits the work and sets the handlers (marked red lines in Figure 5.1), while T_2 and T_3 continue to wait. Now only T_1 can trigger the waiting and is therefore capable of keeping T_2 and T_3 from progressing. This is undesirable as it can lead to deadlocking if by some reason T_1 does not wait and at the very least may lead to unnecessary delay for T_2 and T_3 if T_1 does not wait immediately.

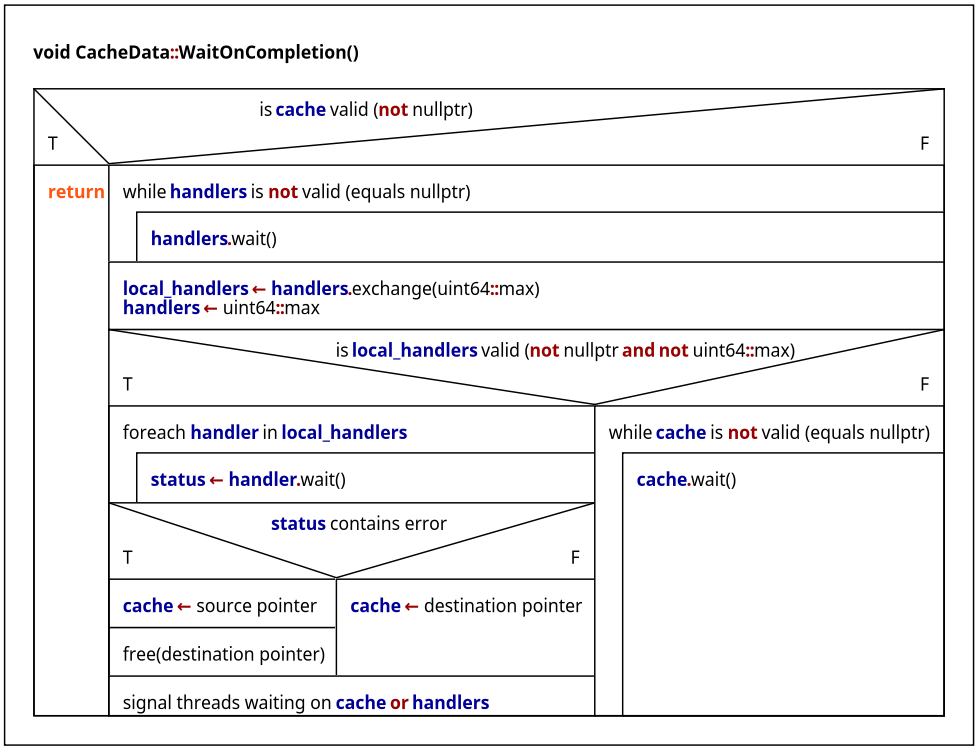

Figure 5.2: `CacheData::WaitOnCompletion` Pseudocode. Final rendition of the implementation for a fair wait function.

To solve this, a different and more complicated order of waiting operations is required. When waiting, the threads now immediately check whether the cache pointer contains a valid value and return if it does, as nothing has to be waited for in this case. Let's take the same example as before to illustrate the second part of the waiting procedure. T_2 and T_3 now both arrive in this latter section as the cache was invalid at the point in time when waiting was called for. They now atomically wait on the handlers pointer to change, instead of doing it the other way around as before. Now when T_1 supplies the handlers, it also uses `std::atomic<T>::notify_one` [19] to wake at least one thread waiting on value change of the handlers pointer, if there are any. Through this the exclusion that was observable in the first implementation is already avoided. If nobody is waiting, then the handlers will be set to a valid pointer and a thread may pass the atomic wait instruction later on. Following this wait, the handlers pointer is atomically exchanged [20] with nullptr, invalidating it. Now each thread again checks whether it has received a valid local pointer to the handlers from the exchange, if it has then the atomic operation guarantees that is now in sole possession of the pointer. The owning thread is tasked with actually waiting. All other threads will now regress and call `CacheData::WaitOnCompletion` again. The solo thread may proceed to wait on the handlers and should update the cache pointer.

Some two additional cases must be considered for the latter implementation to be safe. The wait operation first checks for a valid cache pointer and then waits on the

handlers becoming valid. After processing the handlers, they are deleted and the pointer therefore invalidated. Should the cache pointer now be invalid as well, deadlocks would ensue. Therefore, the thread which exchanged the handlers pointer for a valid local copy must set the cache pointer to a valid value. Should one of the offloaded operations have failed, using the cache pointer is out of question as the datum it references might be invalid. The cache is set to the source address in this case. Secondly, after one thread has exchanged the pointer locally, threads may collect waiting on the handlers to become available. This can happen when the wait on the handlers takes sufficient amount of time during which both handlers and cache pointer are invalid. After waiting, the responsible thread must therefore signal all [21] threads waiting on the handler to continue.

Two types of deadlocks were encountered during testing and have been accounted for. On one hand, it was found that the guarantee of `std::atomic<T>::wait` to only wake up when the value has changed [17] is stronger than the promise of waking up all waiting threads with `std::atomic<T>::notify_all` [21]. The value of the handler pointer may therefore not be exchanged with nullptr which is the value we wait on. As the highest envisioned address requires the lower 52-bits of current 64-bit wide systems [22, p. 120] [23, p. 4-2] setting all bits of a 64-bit-value yields an invalid pointer which is used as the second invalid state possible. The second type was encountered when after creating a `CacheData` instance it was determined this exists in the cache already and dropped. As destruction waits on completion in order to ensure that no further jobs require the memory held, a deadlock would arise from the cache and handler pointers both being null and no handlers ever being set due to the instance being deleted immediately. To circumvent this, the constructor of `CacheData` was modified to point to source memory by default. Only after calling a separate initialization function will `CacheData` replace this with nullptr, therefore readying the instance for multithreaded usage.

5.1.3 Performance Guideline

Atomic operations come with an added performance penalty. No recent studies were found on this, although we assume that the findings of Hermann Schweizer et al. in “Evaluating the Cost of Atomic Operations on Modern Architectures“ [24] still hold true for today’s processor architectures. Due to the inherent cache synchronization mechanisms in place [24, Subsection IV.A.3 Off-Die Access], they observed significant access latency increase depending on whether the atomic variable was located on the local core, on a different core on the same chip or on another socket [24, Fig. 4]. Reducing the cost of atomic accesses would require a less generic implementation, reducing some of the guarantees we give in 4.2. This would allow reducing the amount of atomics required but is outside the scope of this work.

With the distributed locking described in 5.1.1, lock contention should not have a significant impact, although this remains to be tested. In addition to that, passive waiting at the contended section will benefit other threads and might allow overall progress to continue, if the application utilizing the cache has a threading model supporting this. These two factors lead us to classify lock contention as only a minor performance problem.

5.2 Accelerator Usage

After **??** the implementation of `Cache` provided leaves it up to the user to choose a caching and copy method policy which is accomplished through submitting function pointers at initialization of the `Cache`. In 2.5 we configured our system to have separate Nodes for accessing HBM which are assigned a Node-ID by adding eight to the Nodes ID of the Node that physically contains the HBM. Therefore, given Node 3 accesses some datum, the most efficient placement for the copy would be on Node $3 + 8 = 11$. As the `Cache` is intended for multithreaded usage, conserving accelerator resources is important, so that concurrent cache requests complete quickly. To get high per-copy performance while maintaining low usage, the smart-copy method is selected as described in 3.2.3 for larger copies, while small copies will be handled exclusively by the current node. This distinction is made due to the overhead of assigning the current thread to the selected nodes, which is required as Intel DML assigns submissions only to the DSA engine present on the node of the calling thread [7, Section "NUMA support"]. No testing has taken place to evaluate this overhead and determine the most effective threshold.

5.3 Application to Query-driven Prefetching

write this section or consider putting it in evaluation

6 Evaluation

In this chapter we will define our expectations, applying the developed Cache to Query-driven Prefetching. To measure the performance, we adapted code developed by colleagues André Berthold and Anna Bartuschka for evaluating QdP in [25].

6.1 Expectations

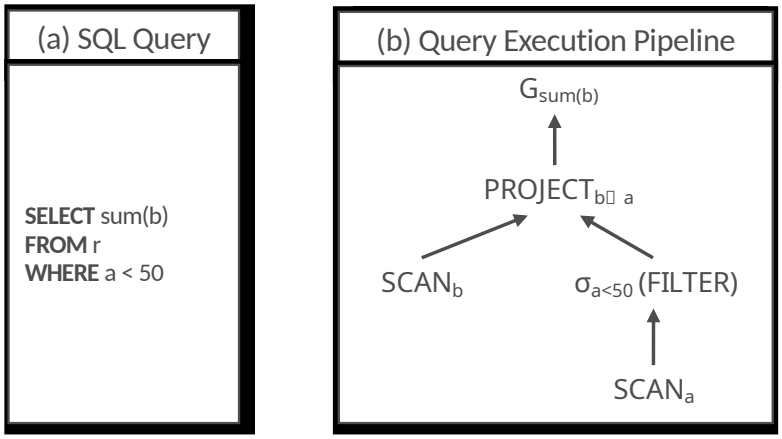

Figure 6.1: Illustration of the benchmarked simple query in (a) and the corresponding pipeline in (b). Taken from [25, Fig. 1].

The benchmark executes a simple query as illustrated in Figure 6.1 which presents a challenging scenario to the cache. As the filter operation applied to `a` is not particularly complex, its execution time can be assumed to be short. Therefore, the Cache has little time during which it must prefetch, which will amplify delays caused by processing overhead in the Cache itself or from submission to the Work Queue. This makes the chosen query suited to stress test the developed solution.

With this difficult scenario, we expect to spend time analysing runtime behaviour of our benchmark in order to optimize the Cache and the way it is applied to the query. Optimizations should yield slight performance improvement over the baseline, using DRAM, and will not reach the theoretical peak, where the data for `b` resides in HBM.

6.2 Observation and Discussion

7 Conclusion And Outlook

write introductory paragraph

7.1 Conclusions

write this section

7.2 Future Work

write this section

- evaluate performance with more complex query
- evaluate impact of lock contention and atomics on performance
- implement direct dsa access to assess gains from using shared work queue
- improve the cache implementation for use cases where data is not static

Glossary

B

BAR

... desc ...

D

DDR-SDRAM

... desc ...

DMR

... desc ...

DSA

... desc ...

DWQ

... desc ...

E

ENQCMD

... desc ...

H

HBM

... desc ...

I

Intel DML

... desc ...

IOMMU

... desc ...

M

MOVDIR64B

... desc ...

N

Node

... desc ...

P

PASID

... desc ...

Q

QdP

... desc ...

S

SWQ

... desc ...

W

WQ

... desc ...

Bibliography

[1] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin and K. Kim, 'Hbm (high bandwidth memory) dram technology and architecture', in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4. DOI: 10.1109/IMW.2017.7939084.

[2] Intel. 'Intel® Xeon® CPU Max Series Product Brief'. (6th Jan. 2023), [Online]. Available: https://www.intel.com/content/www/us/en/content-details/765259/intel-xeon-cpu-max-series-product-brief.html (visited on 18th Jan. 2024).

[3] Intel. 'Intel® Data Streaming Accelerator Architecture Specification'. (16th Sep. 2022), [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671116/intel-data-streaming-accelerator-architecture-specification.html (visited on 15th Nov. 2023).

[4] Intel. 'New Intel® Xeon® Platform Includes Built-In Accelerators for Encryption, Compression, and Data Movement'. (Dec. 2022), [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-12/storage-engines-4th-gen-xeon-brief.pdf (visited on 15th Nov. 2023).

[5] R. K. et al. 'A Quantitative Analysis and Guideline of Data Streaming Accelerator in Intel® 4th Gen Xeon® Scalable Processors'. (May 2023), [Online]. Available: https://arxiv.org/pdf/2305.02480.pdf (visited on 7th Jan. 2024).

[6] Intel. 'Intel® Data Streaming Accelerator User Guide'. (11th Jan. 2023), [Online]. Available: https://www.intel.com/content/www/us/en/content-details/759709/intel-data-streaming-accelerator-user-guide.html (visited on 15th Nov. 2023).

[7] Intel, *Intel Data Mover Library Documentation*, https://intel.github.io/DML/documentation/api_docs/high_level_api.html. (visited on 7th Jan. 2024).

[8] Intel, *Intel IDXD User Space Application*, https://github.com/intel/idxd-config. (visited on 7th Jan. 2024).

[9] J. C. Sam Kuo. 'Implementing High Bandwidth Memory and Intel Xeon Processors Max Series on Lenovo ThinkSystem Servers'. (26th Jun. 2023), [Online]. Available: https://lenovopress.lenovo.com/lp1738.pdf (visited on 21st Jan. 2024).

[10] A. Huang. 'Enabling Intel Data Streaming Accelerator on Lenovo ThinkSystem Servers'. (), [Online]. Available: https://lenovopress.lenovo.com/lp1582.pdf (visited on 18th Apr. 2022).

[11] A. C. Fürst, *Accompanying Thesis Repository*. [Online]. Available: https://git.constantin-fuerst.com/constantin/bachelor-thesis.

[12] Intel. 'Intel® Xeon® CPU Max Series Configuration and Tuning Guide'. (Aug. 2023), [Online]. Available: https://cdrdv2-public.intel.com/787743/354227-intel-xeon-cpu-max-series-configuration-and-tuning-guide-rev3.pdf (visited on 21st Jan. 2024).

[13] Microsoft. 'Allocating Memory from a NUMA Node'. (1st Jul. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/memory/allocating-memory-from-a-numa-node (visited on 28th Jan. 2024).

[14] cppreference.com. 'CPP Reference Entry on std::shared_ptr<T>'. (), [Online]. Available: https://en.cppreference.com/w/cpp/memory/shared_ptr (visited on 17th Jan. 2024).

[15] T. Ku and N. Jung, 'Implementation of Lock-Free shared_ptr and weak_ptr for C++11 multi-thread programming', in *Journal of Korea Game Society*, vol. 21, 28th Feb. 2021, pp. 55–65. DOI: 10.7583/jkgs.2021.21.1.55..

[16] cppreference.com. 'CPP Reference List of Atomic Operations'. (), [Online]. Available: https://en.cppreference.com/w/cpp/thread#Atomic_operations (visited on 18th Jan. 2024).

[17] cppreference.com. 'CPP Reference Entry on std::atomic<T>::wait'. (), [Online]. Available: https://en.cppreference.com/w/cpp/atomic/atomic/wait (visited on 18th Jan. 2024).

[18] T. Rodgers. 'Implementing C++20 atomic waiting in libstdc++'. (6th Dec. 2022), [Online]. Available: https://developers.redhat.com/articles/2022/12/06/implementing-c20-atomic-waiting-libstdc#how_can_we_implement_atomic_waiting_ (visited on 18th Jan. 2024).

[19] cppreference.com. 'CPP Reference Entry on std::atomic<T>::notify_one'. (), [Online]. Available: https://en.cppreference.com/w/cpp/atomic/atomic/notify_one (visited on 18th Jan. 2024).

[20] cppreference.com. 'CPP Reference Entry on std::atomic<T>::exchange'. (), [Online]. Available: https://en.cppreference.com/w/cpp/atomic/atomic/exchange (visited on 18th Jan. 2024).

[21] cppreference.com. 'CPP Reference Entry on std::atomic<T>::notify_all'. (), [Online]. Available: https://en.cppreference.com/w/cpp/atomic/atomic/notify_all (visited on 18th Jan. 2024).

[22] AMD. 'AMD64 Programmer's Manual Volume 2: System Programming'. (Dec. 2016), [Online]. Available: https://support.amd.com/TechDocs/24593.pdf (visited on 18th Jan. 2024).

[23] Intel. 'Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1'. (Dec. 2016), [Online]. Available: https://support.amd.com/TechDocs/24593.pdf (visited on 18th Jan. 2024).

[24] H. Schweizer, M. Besta and T. Hoefler, 'Evaluating the Cost of Atomic Operations on Modern Architectures', in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 445–456. DOI: 10.1109/PACT.2015.24.

[25] A. Berthold, A. Bartuschka, D. Habich, W. Lehner and H. Schirmeier, ‘Towards Query-Driven Prefetching to Optimize Data Pipelines in Heterogeneous Memory Systems’, 2023.