Bachelors Thesis

Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Anatol Constantin Fürst

9th February 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, 9th February 2024

Anatol Constantin Fürst

Abstract

This bachelor's thesis explores the dynamic landscape of heterogeneous memory systems, characterized by advancements in main memory technologies such as Non-Volatile RAM (NVRAM), High Bandwidth Memory (HBM), and Remote Memory. Systems equipped with more than one type of main memory necessitate strategic decisions regarding data placement to take advantage of the properties of the different storage tiers. The responsibility for maintaining optimal data placement falls upon the CPU, resulting in a reduction of available cycles for computational tasks. In response to this challenge, Intel has introduced the Data Streaming Accelerator (DSA), which offloads data operations, offering a potential avenue for enhancing efficiency in data-intensive applications. The primary objective of this thesis is to provide a comprehensive analysis and characterization of the architecture and performance of the DSA, along with its application to a domain-specific prefetching methodology aimed at accelerating database queries within heterogeneous memory systems.

Contents

List of Figures

List of Tables

1 Introduction

The proliferation of various technologies, such as Non-Volatile RAM (NVRAM), High Bandwidth Memory (HBM), and Remote Memory, has ushered in a diverse landscape of systems characterized by varying tiers of main memory. Within these systems, the movement of data across memory classes becomes imperative to leverage the distinct properties offered by the available technologies. Traditionally tasked with managing data locality, the CPU faces an added burden in heterogeneous memory environments, thereby diminishing available processing cycles. To mitigate this strain on the CPU, certain Intel Server Processors now feature the Intel Data Streaming Accelerator (DSA), to which certain data operations may be offloaded [1]. With it, this thesis undertakes the challenge of optimizing data locality on Non Uniform Memory Architectures.

The primary objectives of this thesis are twofold. Firstly, it involves a comprehensive analysis and characterization of the architecture of the Intel DSA. Secondly, the focus extends to the application of DSA in the domain-specific context of Query-driven Prefetching (QdP) to accelerate database queries [2].

This work introduces significant contributions to the field. Notably, the design and implementation of an offloading cache represent a key highlight, providing an interface for leveraging the strengths of tiered storage with minimal integration efforts. Additionally, the thesis includes a detailed examination and analysis of the strengths and weaknesses of the DSA through microbenchmarks. These benchmarks serve as practical guidelines, offering insights for the optimal application of DSA in various scenarios. As of the time of writing, this thesis stands as the first scientific work to extensively evaluate the DSA in a multi-socket system and provide benchmarks for programming through the Intel Data Mover Library (Intel DML). Furthermore, performance for data movement from DDR-SDRAM to High Bandwidth Memory (HBM) using DSA has not yet been evaluated by the scientific community.

The Technical Background chapter furnishes the reader with pertinent background information necessary for understanding the subsequent sections of this work, encompassing High Bandwidth Memory (HBM), Query-driven Prefetching (QdP), and DSA along with its programming interface [3]. Additionally, guidance on system setup and configuration is provided. Subsequently, the Performance Microbenchmarks section analyzes the strengths and weaknesses of the DSA. Methodologies are presented, each benchmark is elaborated upon in detail, and usage guidance is drawn from the results. The following sections, Design and Implementation, elucidate the practical aspects of the work, including the development of the interface and implementation for an offloading cache, shedding light on specific design considerations and implementation challenges. The Evaluation section offers a comprehensive assessment of the implemented solution. In Conclusion, insights gained are reflected upon, and the contributions and results of the preceding chapters are reviewed.

2 Technical Background

This chapter introduces the relevant technologies and concepts for this thesis. The goal of this thesis is to apply the Intel Data Streaming Accelerator to the concept of Query-driven Prefetching, therefore we will familiarize ourselves with both of these terms. We also give background on High Bandwidth Memory, which is a secondary memory technology to the DDR-SDRAM used in current computers.

2.1 High Bandwidth Memory

High Bandwidth Memory is an emerging memory technology that promises an increase in peak bandwidth. It consists of stacked DDR-SDRAM dies [4, p. 1] and is gradually being integrated into server processors, with the Intel® Xeon® Max Series [5] being one recent example. HBM on these systems can be configured in different memory modes, most notably, HBM Flat Mode and HBM Cache Mode [5]. 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 [5].

2.2 Query-driven Prefetching

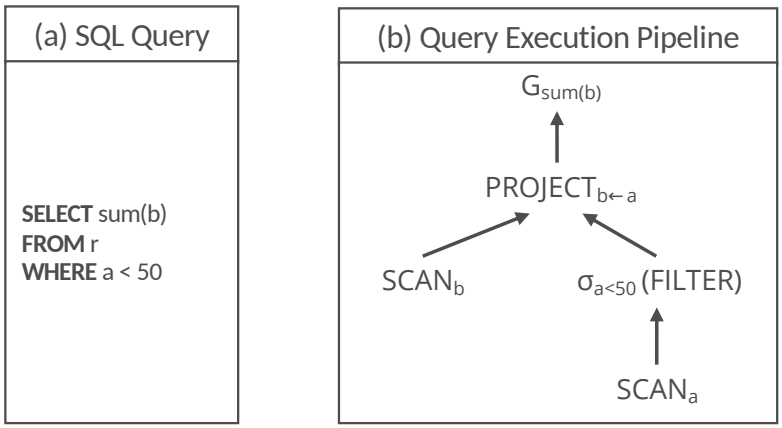


Figure 2.1: Illustration of a simple query in (a) and the corresponding pipeline in (b). [2, Fig. 1]

QdP introduces a targeted strategy for optimizing database performance by intelligently prefetching relevant data. To achieve this, QdP analyses queries, splitting them into distinct sub-tasks, resulting in the so-called query execution plan. An example of a query

and a corresponding plan is depicted in Figure 2.1. From this plan, QdP determines columns in the database used in subsequent tasks. Once identified, the system proactively copies these columns into faster memory during the execution of the pipeline. For the example (Figure 2.1), column `b` is accessed in $SCAN_b$ and $G_{sum(b)}$ and column `a` is only accessed for $SCAN_a$. Therefore, only column `b` will be chosen for prefetching in this scenario. [2]

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. [6, 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' [1, 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

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 [8, 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 2.2. A WQ is accessible through so-called portals, light blue in Figure 2.2, 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

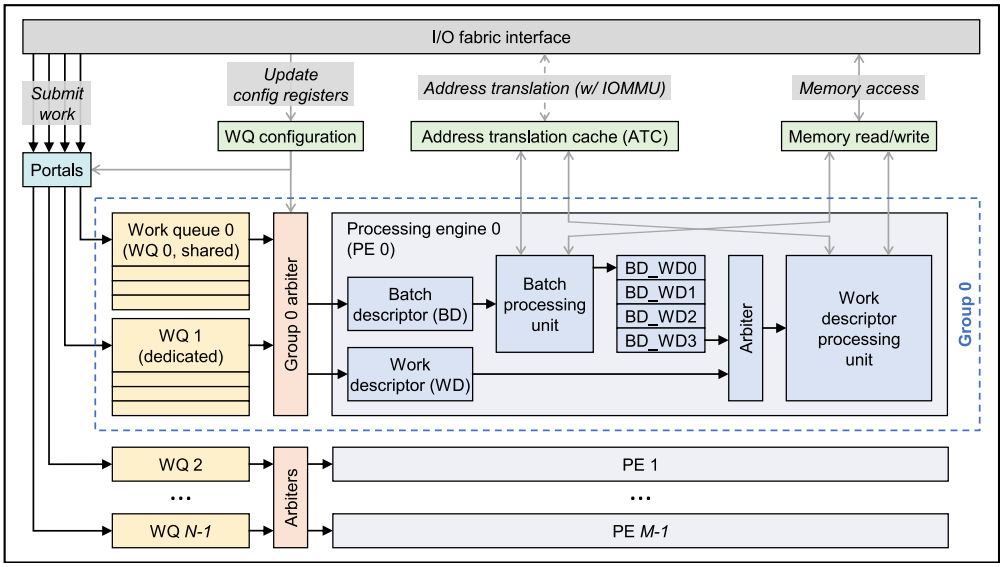


Figure 2.2: Intel Data Streaming Accelerator Internal Architecture. Shows the components that the chip is made up of, how they are connected and which outside components the DSA communicates with. [7, Fig. 1 (a)]

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 [6, 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 [6, 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 [6, Sec. 3.2]. For a batch, the DSA reads the batch descriptor, then fetches all task descriptors from memory and processes them [6, 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 [6, Sec. 2.2, Block on Fault].

Component III, Groups: Groups tie Engines and Work Queues together, indicated by the dotted blue line around the components of Group 0 in Figure 2.2. 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 2.2, 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 2.2. 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 [6, Sec. 3.3.1] or set statically after a process is attached to a DWQ [6, 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' [6, 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. [6, Sec. 3.9]

2.3.2 Software View

Since Linux Kernel 5.10, there exists a driver for the DSA which has no counterpart in the Windows OS-Family [3, 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 [9] 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 2.3. It interacts with the kernel driver, visible in light green and labelled 'IDXD' in Figure 2.3. After successful configuration, each WQ is exposed as a character device through `mmap` of the associated portal [7, Sec. 3.3].

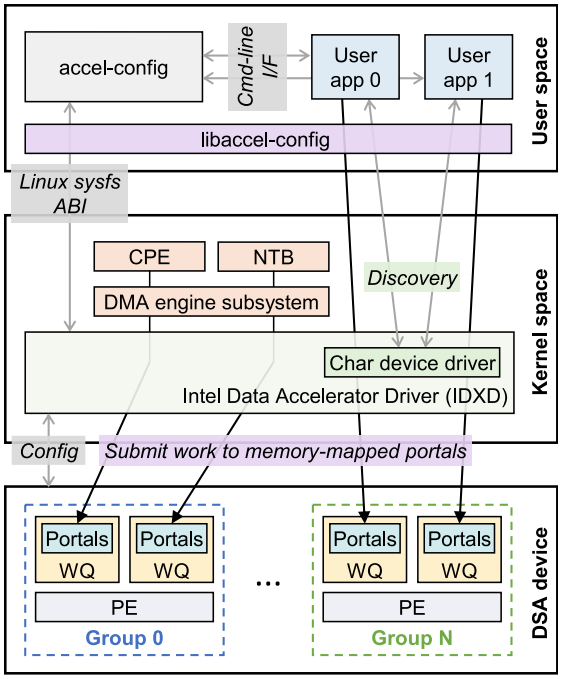


Figure 2.3: Intel Data Streaming Accelerator Software View. Illustrating the software stack and internal interactions from user applications, through the driver to the portal for work submission. [7, Fig. 1 (b)]

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. [3, Sec. Introduction]

2.4 Programming Interface for Intel Data Streaming Accelerator

As mentioned in Section 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.

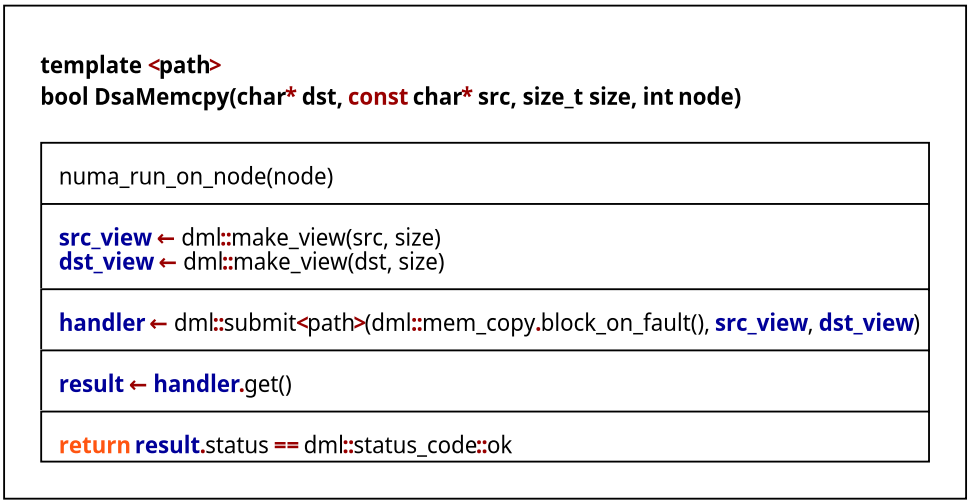


Figure 2.4: Memcpy Implementation Pseudocode. Performs copy operation of a block of memory from source to destination. The DSA executing this copy can be selected with the parameter `node`, and the template parameter `path` elects whether to run on hardware (Intel DSA) or software (CPU).

In the function header of Figure 2.4 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 [3, Sec. Quick Start].

Choosing the engine which carries out the copy might be advantageous for performance, as we can see in Section 3.2.3. With the engine directly tied to the processing node, as observed in Section 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 2.4, where these views are labelled `src_view` and `dst_view`. [3, Sec. High-level C++ API, Make view]

In Figure 2.4, 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. [3, Sec. High-level C++ API, How to Use the Library] [3, 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 [10, Sec. Configuring HBM, Configuring Flat Mode], 'VT-d' to Enabled in BIOS [8, Sec. 2.1] and, if available, 'Limit CPU PA to 46 bits' to Disabled in BIOS [11, p. 5]

2. Use a kernel with IDXD driver support, available from Linux 5.10 or later [3, Sec. Installation] and append the following to the kernel boot parameters in grub config: `intel_iommu=on,sm_on` [11, 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 [11, p. 5]

4. Install `accel-config` from repository [9] or system package manager and inspect the detection of DSA devices through the driver using `accel-config list -i` [11, p. 6]

5. Create DSA configuration file for which we provide an example under `benchmarks/configuration-files/8n1d1e1w.conf` in the accompanying repository [12] that is used for most benchmarks available. Then apply the configuration using `accel-config load-config -c [filename] -e` [8, Fig. 3-9]

6. Inspect the now configured DSA devices using `accel-config list` [11, 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 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 [7]. Therefore, we will perform only a limited amount of benchmarks with the purpose of verifying the figures from [7] and analysing best practices and restrictions for applying DSA to QdP.

3.1 Benchmarking Methodology

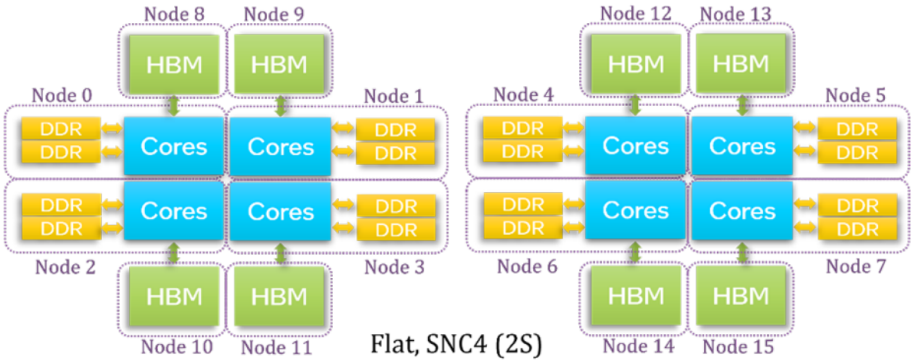


Figure 3.1: Xeon Max Layout [13, Fig. 14] for a 2-Socket System when configured with HBM-Flat. Showing separate Node IDs for manual HBM access and for Cores and DDR-SDRAM.

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 [12].

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. We therefore also do not use '.block_on_fault()', as we did for the memcpy-example in Section 2.4.

Timing in the outer loop may display lower throughput than actual. This is the case, should one of the DSAs participating in a given task finish earlier than the others. We decided to measure the maximum time and therefore minimum throughput for these cases, as we want the benchmarks to represent the peak achievable for distributing one task over multiple engines and not executing multiple tasks of a disjoint set. As a task can only be considered complete when all subtasks are completed, the minimum throughput represents this scenario. This may give an advantage to the peak CPU throughput benchmark we will reference later on, as it does not have this restriction placed upon it.

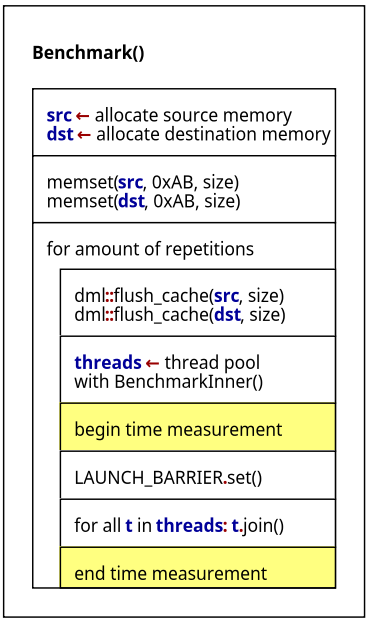


Figure 3.2: Outer Benchmark Procedure Pseudocode. Timing marked with yellow background. Showing preparation of memory locations, clearing of cache entries, timing points and synchronized benchmark launch.

To get accurate results, the benchmark is repeated 10 times. Each iteration is timed from beginning to end, marked by yellow in Figure 3.2. For small task sizes, the iterations complete in a very short amount of time, which can have adverse effects on the results. Therefore, we repeat the code of the inner loop for a configurable amount, virtually extending the duration of a single iteration for these cases.

For all DSAs used in the benchmark, a submission thread executing the inner benchmark routine is spawned. The launch is synchronized by use of a barrier for each iteration. The behaviour in the inner function 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 3.3 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

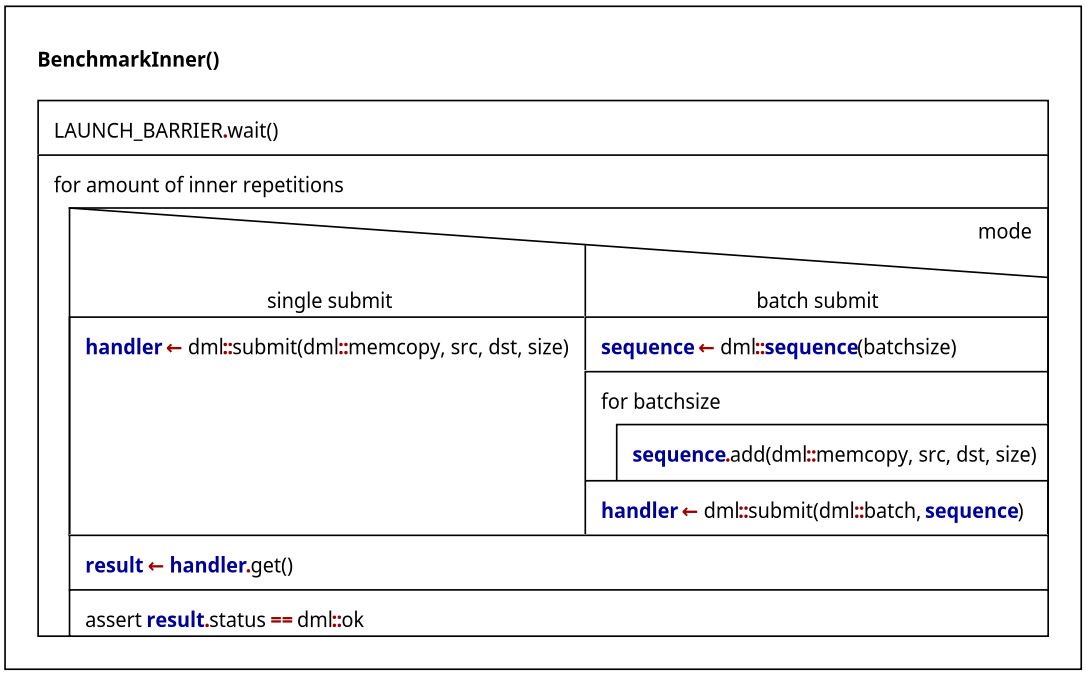


Figure 3.3: Inner Benchmark Procedure Pseudocode. Showing work submission for single and batch submission.

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.

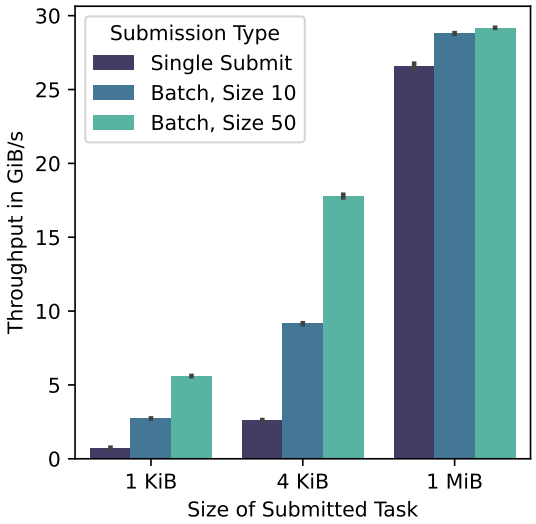


Figure 3.4: Throughput for different Submission Methods and Sizes. Performing a copy with source and destination being node 0, executed by the DSA on node 0. Observable is the submission cost which affects small transfer sizes differently, as there the completion time is lower.

In Figure 3.4 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. Reese Kuper et al. observed that 'SWQ observes lower throughput between 1-8 KB [transfer size]' [7, pp. 6]. We however observe a much higher point of equalization, pointing to additional delays introduced by programming the DSA through Intel DML.

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 [7, p. 5].

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, with the latter representing the core count of one processing sub-node on the test system. We spawn multiple threads, all submitting to one DSA. Furthermore, 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.

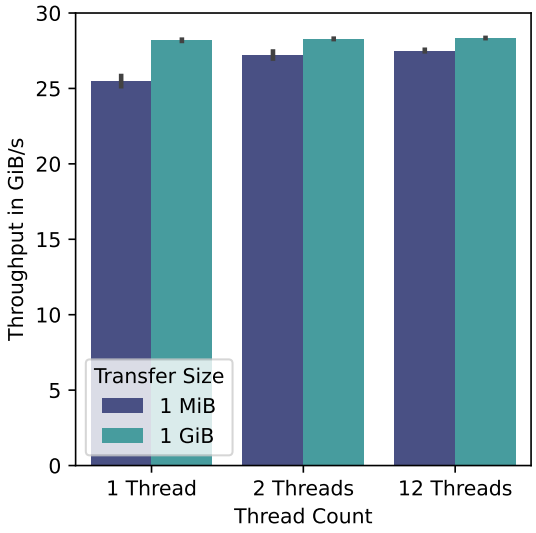


Figure 3.5: Throughput for different Thread Counts and Sizes. Multiple threads submit to the same Shared Work Queue. Performing a copy with source and destination being node 0, executed by the DSA on node 0.

In Figure 3.5, 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 increased with the addition of threads, which we attribute to enhanced queue usage. We ascribe the higher throughput observed with 1 GiB to the submission delay which is incurred more frequently with lower transfer sizes.

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. For each node, the test system is configured with two DIMMs of DDR5-4800.

The naming scheme contains the data rate in Megatransfers per second. We calculate the transfers performed per second. [14]

$$2\ DIMM * \frac{4800\ MT}{s\ *\ DIMM} = 9600\ MT/s$$

The data width of DDR5 is 64 bit. We calculate the amount of Bytes per Transfer. [14]

$$\frac{64b}{8b/B}\ /\ Transfer = 8B\ /\ Transfer$$

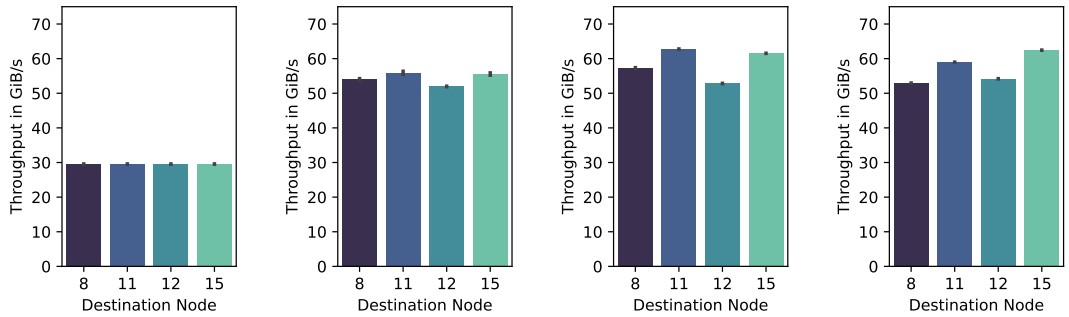
Using the results from the previous calculations, we are now able to calculate the theoretical peak throughput speed per Node on our test system.

$$9600\ MT/s * 8B/T = 75\ GiB/s$$

We conclude that to achieve peak speeds, a copy task has to be split across multiple DSAs. Different methods of splitting will be evaluated. Given that our system consists of multiple sockets, communication crossing between sockets could introduce latency and bandwidth disadvantages [15].

To determine the optimal amount of DSAs, we will measure throughput for one, two, four, and eight participating in the copy operations. We name the utilization of two DSAs 'Push-Pull', as with two accelerators, we utilise the ones found on data source and destination node. As eight DSAs is the maximum available on our system, this configuration will be referred to as 'brute-force'.

For this benchmark, we transfer 1 Gibibyte of data from node 0 to the destination node. 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 farthest intra-socket operation benchmarked. Nodes 12 and 15 lie diagonally on the second socket's CPU, making them representative of inter-socket transfer operations.



(a) One DSA: DSA on source Node.

(b) Two DSAs, or 'Push-Pull': using the DSA on source and destination Node except intra-node using the on-node and one off-node.

(c) Four DSAs: using four on-socket DSA for intra-socket and two on each socket for inter-socket.

(d) Eight DSAs or 'Brute-Force': using all available DSA, irrespective of source and destination locations.

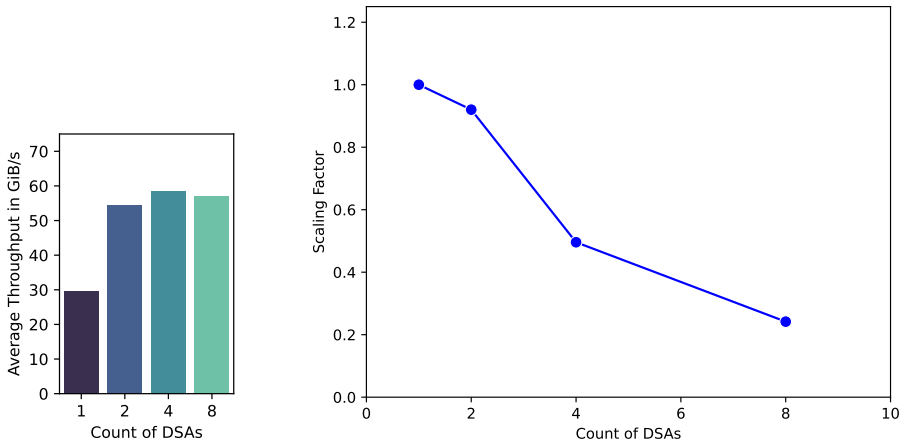
Figure 3.6: Copy from Node 0 to the destination Node specified on the x-axis. Shows peak throughput achievable with DSA for different load balancing techniques.

We begin by examining the common behaviour of load balancing techniques depicted in Figure 3.6. The real-world peak throughput approaches nearly 64 GiB/s, aligning with the maximum achievable with the CPU, as demonstrated in Section 3.2.4. In Figure 3.6a, a notable hard bandwidth limit is observed, just below the 30 GiB/s mark, reinforcing

what was encountered in Section 3.2.1: a single DSA is constrained by I/O-Fabric limitations.

Unexpected throughput differences are evident for all configurations, except the bandwidth-bound single DSA. Notably, NUMA-Node (Node) 8 performs worse than copying to Node 11. As Node 8 serves as the HBM accessor for the data source Node, it should have the shortest data path. This suggests that the DSA may suffer from sharing parts of the data path for reading and writing. Another interesting observation is that, contrary to our assumption, the physically more distant Node 15 achieves higher throughput than the closer Node 12. We lack an explanation for this anomaly and will further examine this behaviour in the analysis of the CPU throughput results in Section 3.2.4.

For the results of the Brute-Force approach illustrated in Figure 3.6d, we observe peak speeds when copying across sockets from Node 0 to Node 15. This contradicts our assumption that peak bandwidth would be limited by the interconnect. However, for intra-node copies, there is an observable penalty for using the off-socket DSAs. We will analyse this behaviour by comparing the different benchmarked configurations and summarize our findings on scalability.



(a) Average Throughput for different amounts of participating DSA.

(b) Scaling Factor for different amounts of participating DSA. Determined by formula $\frac{Throughput}{Basline\ Throughput} * \frac{1}{Utilization\ Factor}$ with the baseline being Throughput for 1 DSA and the utilization factor representing the factor of the amount of DSAs being used over the baseline.

Figure 3.7: Scalability Analysis for different amounts of participating DSAs. Displays the average throughput and the derived scaling factor. Shows that, although the throughput does increase with adding more accelerators, beyond two, the gained speed drops significantly. Calculated over the results from Figure 3.6 and therefore applies to copies from DDR-SDRAM to HBM.

When comparing the Brute-Force approach with Push-Pull in Figure 3.6b, performance decreases by utilizing four times more resources over a longer duration. As shown in

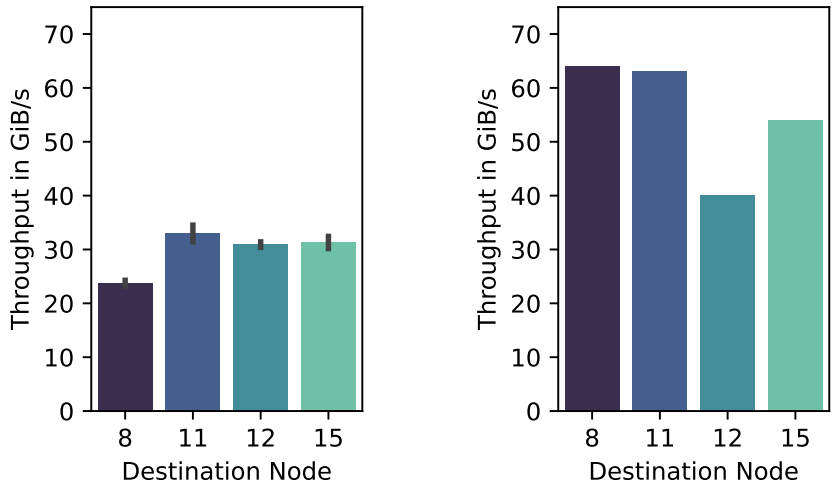
Figure 3.7b, using Brute-Force still leads to a slight increase in overall throughput, although far from scaling linearly. Therefore, we conclude that, although data movement across the interconnect incurs additional cost, no hard bandwidth limit is observable.

From the average throughput and scaling factors in Figure 3.7, it becomes evident that splitting tasks over more than two DSAs yields only marginal gains. This could be due to increased congestion of the overall interconnect, however, as no hard limit is encountered, this is not a definitive answer.

The choice of a load balancing method is not trivial. If peak throughput of one task is of relevance, Brute-Force for inter-socket and four DSAs for intra-socket operation result in the fastest transfers. At the same time, these cause high system utilization, making them unsuitable for situations where multiple tasks may be submitted. For these cases, Push-Pull achieves performance close to the real-world peak while also not wasting resources due to poor scaling — see Figure 3.7b.

3.2.4 Data Movement using CPU

For evaluating CPU copy performance we use the benchmark code from the subsequent Section 3.2.3, selecting the software instead of hardware execution path (see Section 2.3.2). Colleagues performed extensive benchmarking of the peak throughput on CPU for the test system [16], from which we will present results as well. We compare expectations and results from the previous Section with the measurements.



(a) DML code for allnodes running on software path.

(b) Colleague's CPU peak throughput benchmark [16] results.

Figure 3.8: Throughput from DDR-SDRAM to HBM on CPU. Copying from Node 0 to the destination Node specified on the x-axis.

As evident from Figure 3.8a, the observed throughput of software path is less than half of the theoretical bandwidth. Therefore, software path is to be treated as a compatibility

measure, and not for providing high performance data copy operations. As the sole benchmark, the software path however performs as expected for transfers to Nodes 12 and 15, with the latter performing worse. Taking the layout from Figure 3.1, back in the previous Section, we assumed that Node 12 would outperform Node 15 due to lower physical distance. This assumption was invalidated, making the result for CPU in this case unexpected.

In Figure 3.8b, peak throughput is achieved for intra-node operation. This validates the assumption that there is a cost for communicating across sockets, which was not as directly observable with the DSA. The same disadvantage for Node 12, as seen in Section 3.2.3 can be observed in Figure 3.8b. As the results from software path do not exhibit this, the anomaly seems to only occur in bandwidth-saturating scenarios.

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 and otherwise use the CPU.
- Section 3.2.2 assures that access from multiple threads does not negatively affect the performance when using Shared Work Queue for work submission. Due to the lack of Dedicated Work Queue support, we have no data to determine the cost of submission to the SWQ.
- In 3.2.3, we found that using more than two DSAs results in only marginal gains. The choice of a load balancer therefore is the Push-Pull configuration, as it achieves fair throughput with low utilization.

Once again, we refer to Figures 3.6 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. In any case, DSA performs similar to the CPU, demonstrating potential for faster data movement while simultaneously freeing up cycles for other tasks.

We discovered an anomaly for Node 12 for which we did not find an explanation. As the behaviour is also exhibited by the CPU, discovering the root issue falls outside the scope of this work.

Scaling was found to be less than linear. No conclusive answer was given for this either. We assumed that this happens due to increased congestion of the interconnect.

Even though we could not find an explanation for all measurements, this chapter still gives insight into the performance of the DSA, its strengths and its weaknesses. It provides data-driven guidance on a complex architecture, helping to find the optimum for applying the DSA to our expected and possibly different workloads.

4 Design

In this chapter we design a class interface for use as a general purpose cache. We will present challenges and then detail the solutions employed to face them, finalizing the architecture with each step. Details on the implementation of this blueprint will be omitted, as we discuss a selection of relevant aspects in Chapter 5. We also shortly touch the subject of DSA usage.

4.1 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.

4.1.1 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 4.2. 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.

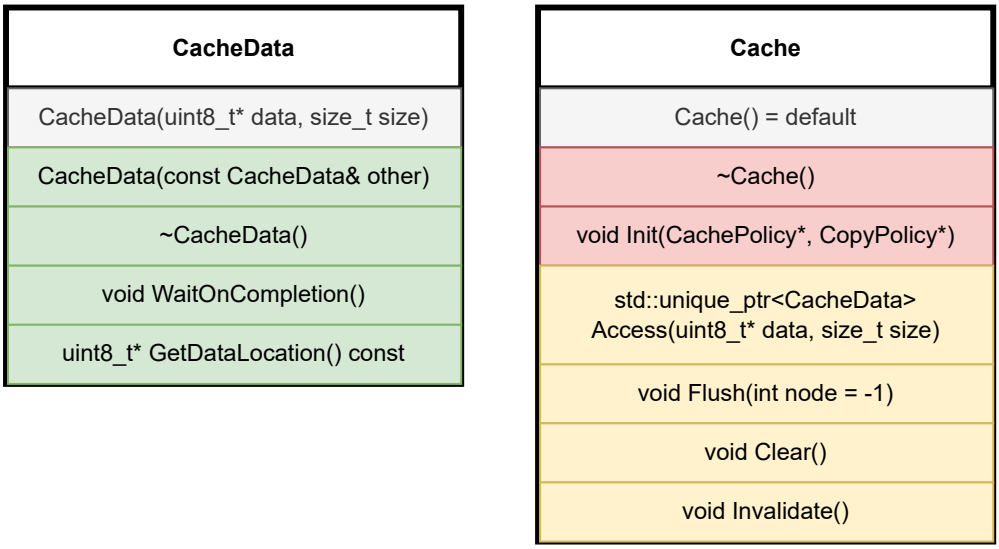


Figure 4.1: Public Interface of `CacheData` and `Cache` Classes. Colour coding for thread safety. Grey denotes impossibility for threaded access. Green indicates full safety guarantees only relying on atomics to achieve this. Yellow may use locking but is still safe for use. Red must be called from a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

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.

4.1.2 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.

4.1.3 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.

4.1.4 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 [17].

4.2 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 [3]. 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 Section 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. Finally, we apply the cache to Query-driven Prefetching, detailing the policies mentioned in Section 4.2 and presenting solutions for the challenges encountered.

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 4.1 we decided to keep elements in the cache until forced by memory pressure to remove them. Secondly in Section 4.1.2 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

Throughout this section we will use the term 'handler', which was coined by Intel DML, referring to an object associated with an operation on the accelerator. Through it, the state of a task may be queried, making the handler our connection to the asynchronously executed task. As we may split up one single copy into multiple distinct tasks for submission to multiple DSAs, `CacheData` internally contains a vector of multiple of these handlers.

The choice made in 4.1.2 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 [18], 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 4.1.2, threads need to coordinate which one performs the actual waiting. To avoid queuing multiple copies of the same task, the task must be added to the cache state before submission. These two aspects necessitate modifying the handlers atomically. We therefore use an atomic pointer in `CacheData` to allow safe exchange and waiting on modification.

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 [19] 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 atomic integer is either incremented or decremented using atomic fetch sub and add operations 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.

Due to the possibility of access by multiple threads, the implementation of `CacheData::WaitOnCompletion` proved to be challenging. In the first implementation, a thread would check if the handlers are available and atomically wait [20] on a value change from `nullptr`, if they are not. 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.

To illustrate this, an exemplary scenario is used, as seen in the sequence diagram Figure 5.1. Assume that three threads T_1, T_2 and T_3 wish to access the same resource.

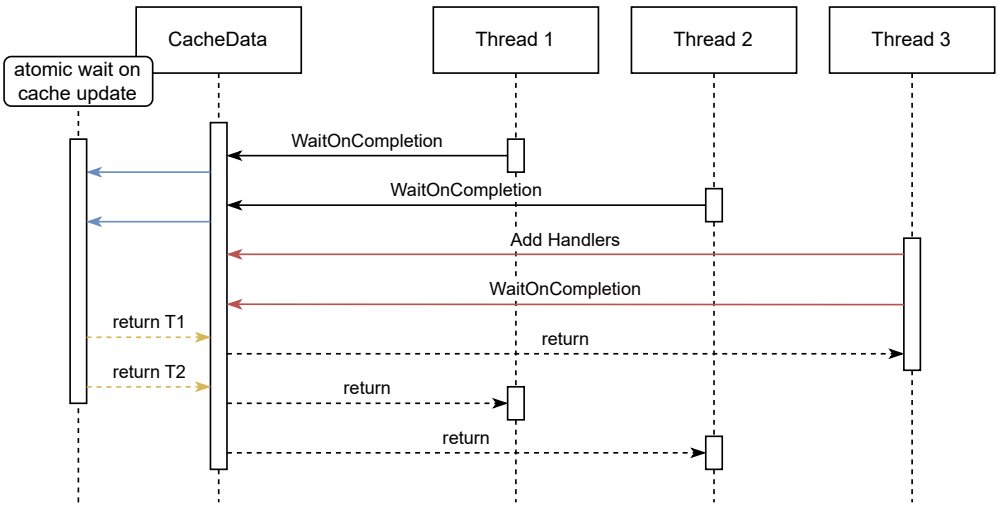


Figure 5.1: Sequence for Blocking Scenario. Observable in first draft implementation. Scenario where T_1 performed first access to a datum followed T_2 and T_3. Then T_1 holds the handlers exclusively, leading to the other threads having to wait for T_1 to perform the work submission and waiting before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

T_1 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). 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. Therefore, 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.

As a solution for this, a more intricate implementation 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. We will use the same example as before to illustrate the second part of the waiting procedure. Both T_2 and T_3 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` [21] 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 [22] with `nullptr`, invalidating it. Each thread again checks whether it has received a valid local pointer to

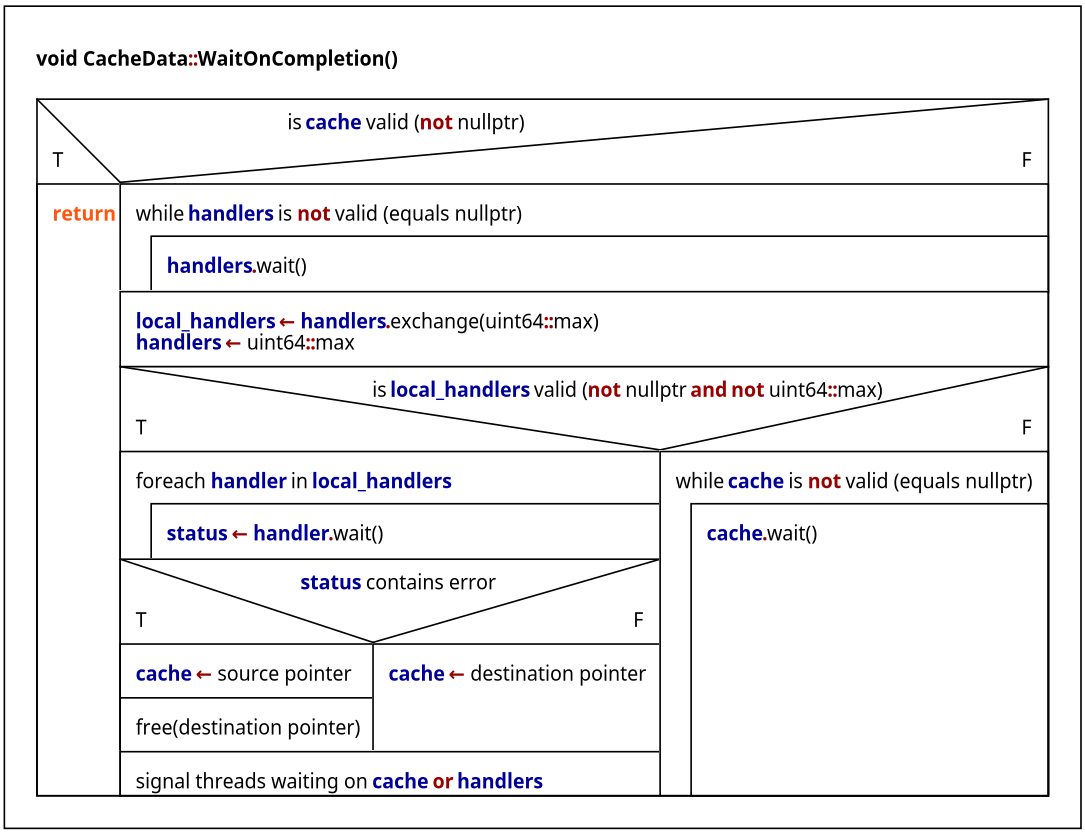


Figure 5.2: `CacheData::WaitOnCompletion` Pseudocode. Final rendition of the implementation for a fair wait function.

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.

Additional cases must be considered for the latter implementation to be safe and free of deadlocks. We will now discuss these edge cases and their resolution.

5.1.2.1 Initial Invalid State

We previously mentioned the possibly problematic situation where both the cache pointer and the handlers are not yet available for an instance in `CacheData`. This situation is avoided explicitly by the implementation due to waiting on the handlers being atomically updated from `nullptr` to valid. When the handlers will be set in the future by the thread calling `Cache::Access` first, progress is guaranteed.

5.1.2.2 Invalid State on Immediate Destruction

The previous Section discussed the initial invalid state and noted that, as long as the handlers will be set in the future, progress is guaranteed. We now discuss the situation where handlers will not be set. This situation is encountered when a memory region is accessed by threads T_1 and T_2 concurrently. One will win the data race to add the entry to the cache state, we choose T_1. T_2 then must follow Section 4.1.2 and return the entry already present in cache state. Therefore, T_2 has to destroy the `CacheData` instance it created previously.

The destructor of `CacheData` waits on operation completion in order to ensure that no running jobs require the cache memory region, before deallocating it. This necessitates usability of `CacheData::WaitOnCompletion` for the case of immediate destruction. As the instance of `CacheData` is destroyed immediately, no tasks will be submitted to the DSA and therefore handlers never become available, leading to deadlock on destruction.

To circumvent this deadlock, the initial state of `CacheData` was modified to be safe for deletion. An initialization function was added to `CacheData`, which is required to be called when the instance is to be used.

5.1.2.3 Invalid State on Operation Failure

`CacheData::WaitOnCompletion` first checks for a valid cache pointer and then waits on the handlers becoming valid. To process the handlers, the global atomic pointer is read into a local copy and then set to `nullptr` using `std::atomic<T>::exchange`. During evaluation of the handlers completion states, an unsuccessful operation may be found. In this case, the cache memory region remains invalid and may therefore not be used. In this case, both the handlers and the cache pointer will be `nullptr`. This results in an invalid state, like the one discussed in Section 5.1.2.1.

In this invalid state, progress is not guaranteed by the measures set forth to handle the initial invalidity. The cache is still `nullptr` and as the handlers have already been set and processed, they will also be `nullptr` without the chance of them ever becoming valid.

Edge case handling is introduced and the cache pointer is set to the source address, providing validity.

5.1.2.4 Locally Invalid State due to Race Condition

The guarantee of `std::atomic<T>::wait` to only wake up when the value has changed [20] was found to be stronger than the promise of waking up all waiting threads with `std::atomic<T>::notify_all` [23].

As visible in Figure 5.2, we wait while the handlers-pointer is `nullptr`, if the cache pointer is invalid. To exemplify we use the following scenario. Both T_1 and T_2 call `CacheData::WaitOnCompletion`, with T_1 preceding T_2. T_1 exchanges the global handlers-pointer with `nullptr`, invalidating it. Before T_1 can check the status of the handlers and update the cache pointer, T_2 sees an invalid cache pointer and then waits for the handlers becoming available.

This has again caused a similar state of invalidity as the previous two Sections handled. As the handlers will not become available again due to being cleared by T_1, the second consumer, T_2, will now wait indefinitely. This missed update is commonly referred to as 'ABA-Problem' for which multiple solutions exist.

One could use double-width atomic operations and introduce a counter which would allow resetting the pointer back to null while setting a flag indicating the exchange took place. The handlers-pointer would then be contained in a struct with this flag, allowing exchange with a composite of `nullptr` and flag-set. Other threads then would then wait on the struct changing from `nullptr` and flag-unset, allowing them to pass if either the flag is set or the handlers have become non-null. As standard C++ does not yet support the required operations, we chose to avoid the missed update differently. [24]

The chosen solution for this is to not exchange the handlers-pointer with `nullptr` but with a second invalid value. We must determine a secondary invalid pointer for use in the exchange. Therefore, we introduce a new attribute, of the same type as the one pointed to by the handlers-pointer, to `Cache`. The `Cache` then shares it with each instance of `CacheData`, where it is then used in `CacheData::WaitOnCompletion`.

This secondary value allows T_2 to pass the wait, then perform the exchange of handlers itself. T_2 then checks the local copy of the handlers-pointer for validity. The invalid state now includes both `nullptr` and the secondary invalid pointer chosen. With this, the deadlock is avoided and T_2 will wait for T_1 completing the processing of the handlers.

5.2 Application to Query-driven Prefetching

Applying the `Cache` to QdP is a straightforward process. We adapted the benchmarking code developed by Anna Bartuschka and André Berthold [2], invoking Cache::Access for both prefetching and cache access. Due to the high amount of smaller submissions, we decided to forego splitting of tasks unto multiple DSA and instead distribute the copy tasks per thread in round-robin fashion to all available. This causes less delay due to submission cost which, as shown in Section 3.2.1, rises with smaller tasks. The cache location is fixed to Node 8, the HBM accessor of Node 0 to which the application will be bound and therefore exclusively run on.

During the performance analysis of the developed `Cache`, we discovered that Intel DML does not utilize interrupt-based completion signaling (Section 2.3.1.3), but instead employs busy-waiting on the completion descriptor being updated. Given that this busy waiting incurs CPU cycles, waiting on task completion is deemed impractical, necessitating code modifications. We extended `CacheData` and Cache to incorporate support for weak waiting. By introducing a flag configurable in `Cache`, all instances of `CacheData` created via `Cache::Access` will check only once whether the DSA has completed processing `Cache` operation, and if not, return without updating the cache-pointer. Consequently, calls to `CacheData::GetDataLocation` may return `nullptr` even after waiting, placing the responsibility on the user to access the data through its source location. For applications prioritizing latency, `Cache::Access` offers the option for weak access. When activated, the function returns only existing instances of `CacheData`, thereby avoiding work submission to the DSA if the address has not been previously

cached or was flushed since the last access. Using these two options, we can avoid work submission and busy waiting where access latency is paramount.

Additionally, we observed inefficiencies stemming from page fault handling. Task execution time increases when page faults are handled by the DSA, leading to cache misses. Consequently, our execution time becomes bound to that of DDR-SDRAM, as misses prompt a fallback to the data's source location. When page faults are handled by the CPU during allocation, these misses are avoided. However, the execution time of the first data access through the `Cache` significantly increases due to page fault handling. One potential solution entails bypassing the system's memory management by allocating a large memory block and implementing a custom memory management scheme. As memory allocation is a complex topic, we opted to delegate this responsibility to the user by mandating the provision of new- and free-like functions akin to the policy functions utilized for determining placement and task distribution. Consequently, the benchmark can pre-allocate the required memory blocks, trigger page mapping, and subsequently pass these regions to the `Cache`.

6 Evaluation

In this chapter we will define our expectations, applying the developed Cache to Query-driven Prefetching, and then evaluate the observed results. The code used is described in more detail in Section 5.2.

6.1 Benchmarked Task

The benchmark executes a simple query as illustrated in Figure 2.1. We will from hereinafter use notations $SCAN_a$ for the pipeline that performs scan and subsequently filter on column `a`, $SCAN_b$ for the pipeline that prefetches column `b` and $AGGREGATE$ for the projection and final summation step. We use a column size of 4 GiB. The work is divided over multiple groups and each group spawns threads for each pipeline step. For a fair comparison, each tested configuration uses 64 Threads for the first stage ($SCAN_a$ and $SCAN_b$) and 32 for the second stage ($AGGREGATE$), while being restricted to run on Node 0 through pinning. For configurations not performing prefetching, $SCAN_b$ is not executed. We measure the times spent in each pipeline, cache hit percentage and total processing time.

Pipelines $SCAN_a$ and $SCAN_b$ execute concurrently, completing their workload and then signalling $AGGREGATE$, which then finalizes the operation. With the goal of improving cache hit rate, we decided to loosen this restriction and let $SCAN_b$ work freely, only synchronizing $SCAN_a$ with $AGGREGATE$. Work is therefore submitted to the DSA as frequently as possible, thereby hopefully completing each caching operation for a chunk of `b` before $SCAN_a$ finishes processing the associated chunk of `a`.

6.2 Expectations

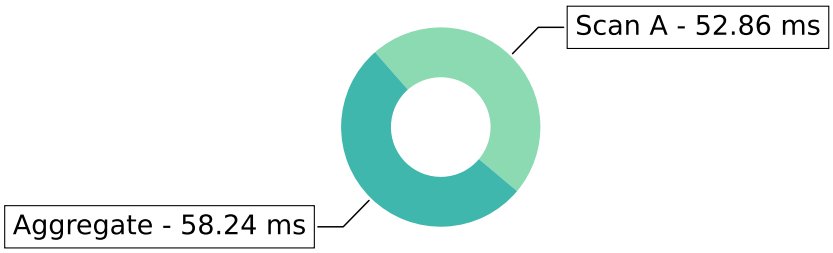
The simple query presents a challenging scenario to the `Cache`. As the filter operation applied to column `a` is not particularly complex, its execution time can be assumed to be short. Therefore, the `Cache` has little time during which it must prefetch, which will amplify delays caused by processing overhead in the `Cache` or during accelerator offload. Additionally, it can be assumed that the task is memory bound. As the prefetching of `b` in $SCAN_b$ and the load and subsequent filter of `a` in $SCAN_a$ will execute in parallel, caching therefore directly reduces the memory bandwidth available to $SCAN_a$, when both columns are located on the same Node.

Due to the challenges posed by sharing memory bandwidth we will benchmark prefetching in two configurations. The first will find both columns `a` and `b` located on the same Node. We expect to demonstrate the memory bottleneck in this situation by execution time of $SCAN_a$ rising by the amount of time spent prefetching in $SCAN_b$. The second

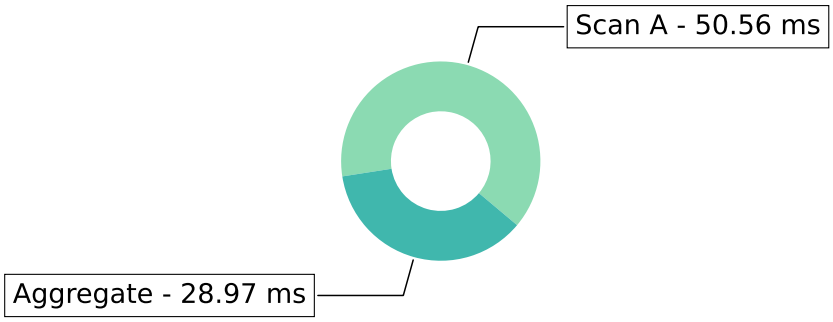
setup will see the columns distributed over two Nodes, still DDR-SDRAM, however. In this configuration $SCAN_a$ should only suffer from the additional threads performing the prefetching.

6.3 Observations

In this section we will present our findings from applying the `Cache` developed in Chatpers 4 and 5 to QdP. We begin by presenting the results without prefetching, representing the upper and lower boundaries respectively.



(a) Columns `a` and `b` located on the same DDR-SDRAM Node.



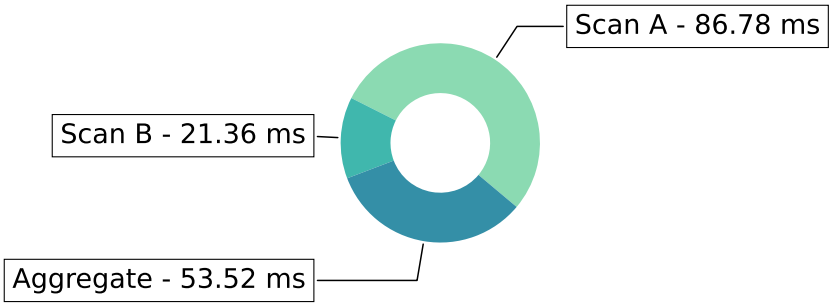
(b) Column `a` located in DDR-SDRAM and `b` in HBM.

Figure 6.1: Time spent on functions $SCAN_a$ and $AGGREGATE$ without prefetching for different locations of column `b`. Figure (a) represents the lower boundary by using only DDR-SDRAM, while Figure (b) simulates perfect caching by storing column `b` in HBM during benchmark setup.

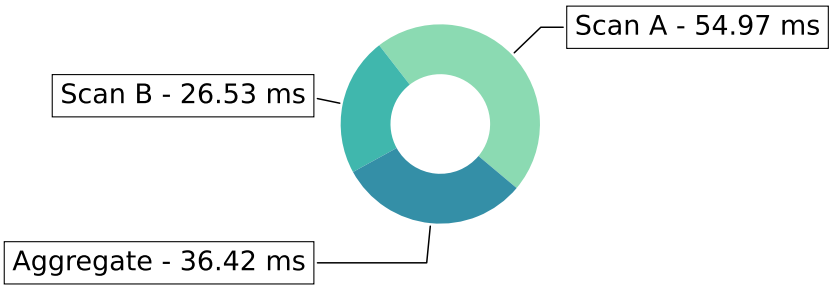
Our baseline will be the performance achieved with both columns `a` and `b` located in DDR-SDRAM and no prefetching. The upper limit will be represented by measuring the scenario where `b` is already located in HBM at the start of the benchmark, simulating prefetching with no overhead or delay.

- Fig 6.2b aggr for prefetch theoretically at level of hbm but due to more concurrent workload aggr gets slowed down too
- Fig 6.2a scana increase due to sharing bandwidth reasonable, aggr seems a bit unreasonable

consider benchmarking only with one group and lowering wl size accordingly to reduce effects of overlap-



(a) Prefetching with columns `a` and `b` located on the same DDR-SDRAM Node.



(b) Prefetching with columns `a` and `b` located on different DDR-SDRAM Nodes.

Figure 6.2: Time spent on functions $SCAN_a$, $SCAN_b$ and $AGGREGATE$ with prefetching. Operations $SCAN_a$ and $SCAN_b$ execute concurrently. Figure (a) shows bandwidth limitation as time for $SCAN_a$ increases drastically due to the copying of column `b` to HBM taking place in parallel. For Figure (b), the columns are located on different Nodes, thereby the $SCAN$-operations do not compete for bandwidth.

Configuration	Speedup	Cache Hitrate
DDR-SDRAM (Baseline)	x1.00	—
HBM (Upper Limit)	x1.41	—
Prefetching	x0.82	89.38 %
Prefetching, Distributed Columns	x1.23	93.20 %

Table 6.1: Table showing Speedup for different QdP Configurations over DDR-SDRAM. Result for DDR-SDRAM serves as baseline while HBM presents the upper boundary achievable with perfect prefetching. Prefetching was performed with the same parameters and data locations as DDR-SDRAM, caching on Node 8 (HBM accessor for the executing Node 0). Prefetching with Distributed Columns had columns `a` and `b` located on different Nodes.

6.4 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

memory pressure

... desc ...

MOVDIR64B

... desc ...

N

Node

... desc ...

P

PASID

... desc ...

Q

QdP

... desc ...

S

SWQ

... desc ...

W

WQ

... desc ...

Bibliography

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

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

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

[4] 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.

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

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

[7] R. Kuper, I. Jeong, Y. Yuan, J. Hu, R. Wang, N. Ranganathan and N. S. Kim, 'A Quantitative Analysis and Guideline of Data Streaming Accelerator in Intel® 4th Gen Xeon® Scalable Processors', May 2023. DOI: 10.48550/arXiv.2305.02480.

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

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

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

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

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

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

[14] Kingston, *DDR5 memory standard: An introduction to the next generation of DRAM module technology*, https://www.kingston.com/en/blog/pc-performance/ddr5-overview, Jan. 2024. (visited on 4th Feb. 2024).

[15] A. Thune, S.-A. Reinemo, T. Skeie and X. Cai, 'Detailed modeling of heterogeneous and contention-constrained point-to-point mpi communication', *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1580–1593, 2023. DOI: 10.1109/TPDS.2023.3253881.

[16] A. Berthold and A. Bartuschka, *Throughput Benchmarks for CPU*, personal communication, 2023.

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

[18] cppreference.com, *CPP Reference Entry on std::shared_ptr<T>*, https://en.cppreference.com/w/cpp/memory/shared_ptr. (visited on 17th Jan. 2024).

[19] 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..

[20] cppreference.com, *CPP Reference Entry on std::atomic<T>::wait*, https://en.cppreference.com/w/cpp/atomic/atomic/wait. (visited on 18th Jan. 2024).

[21] cppreference.com, *CPP Reference Entry on std::atomic<T>::notify_one*, https://en.cppreference.com/w/cpp/atomic/atomic/notify_one. (visited on 18th Jan. 2024).

[22] cppreference.com, *CPP Reference Entry on std::atomic<T>::exchange*, https://en.cppreference.com/w/cpp/atomic/atomic/exchange. (visited on 18th Jan. 2024).

[23] cppreference.com, *CPP Reference Entry on std::atomic<T>::notify_all*, https://en.cppreference.com/w/cpp/atomic/atomic/notify_all. (visited on 18th Jan. 2024).

[24] T. Doumler, *DWCAS in C++*, https://timur.audio/dwcas-in-c, 31st Mar. 2022. (visited on 7th Feb. 2024).