Bachelor Thesis

Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Anatol Constantin Fürst

19th 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 bachelor 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, 19th February 2024

Anatol Constantin Fürst

Abstract

This bachelor thesis explores data locality in heterogeneous memory systems, characterized by advancements in main memory technologies such as Non-Volatile RAM and High Bandwidth Memory. Systems equipped with more than one type of main memory or employing a Non-Uniform Memory Architecture necessitate strategic decisions regarding data placement to take advantage of the properties of the different storage tiers. 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. We contribute a versatile library, capable of performing caching, data replication and prefetching asynchronously, accelerated by the DSA.

Contents

List of Figures

List of Tables

1 Introduction

The proliferation of various technologies, such as Non-Volatile RAM (NVRAM) and HBM, has ushered in a diverse landscape of systems characterized by varying tiers of main memory. Extending traditional Non-Uniform Memory Architecture (NUMA), these systems necessitate the movement of data across memory classes and locations to leverage the distinct properties offered by the available technologies. The responsibility for maintaining optimal data placement falls upon the CPU, resulting in a reduction of available cycles for computational tasks. To mitigate this strain, certain current-generation Intel server processors feature the Intel Data Streaming Accelerator (DSA), to which certain data operations may be offloaded [1]. This work undertakes the challenge of optimizing data locality in heterogeneous memory architectures, utilizing the DSA.

The primary objective of this thesis is 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 QdP to accelerate database queries [2].

We introduce 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. The development efforts resulted in an architecture applicable to any use case requiring NUMA-aware data movement with offloading support to the DSA. 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. To our knowledge, this thesis stands as the first scientific work to extensively evaluate the DSA in a multi-socket system, provide benchmarks for programming through the Intel Data Mover Library (Intel DML), and evaluate performance for data movement from DDR-SDRAM to HBM.

We begin the work by furnishing the reader with pertinent technical information necessary to understand the subsequent sections of this work in Chapter 2. Background is given for HBM and QdP, followed by a detailed account of the DSAs architecture along with an available programming interface. Additionally, guidance on system setup and configuration is provided. Subsequently, Chapter 3 analyses the strengths and weaknesses of the DSA through microbenchmarks. Each benchmark is elaborated upon in detail, and usage guidance is drawn from the results. Chapters 4 and 5 elucidate the practical aspects of the work, including the development of the interface and implementation of the cache, shedding light on specific design considerations and implementation challenges. We comprehensively assess the implemented solution by providing concrete data on gains for an exemplary database query in Chapter 6. Finally, Chapter 7 reflects insights gained, and presents a review of the contributions and results of the preceding chapters.

2 Technical Background

This chapter introduces the technologies and concepts, relevant for the understanding of this thesis. With the goal being to apply the Intel Data Streaming Accelerator to the concept of Query-driven Prefetching, we will familiarize ourselves with both. We also give background on High Bandwidth Memory, which is a secondary memory technology to the DDR-SDRAM used in current computers. This chapter also contains an overview of the Intel Data Mover Library, used to interact with the DSA.

2.1 High Bandwidth Memory

High Bandwidth Memory is an emerging technology that promises an increase in peak bandwidth. As visible in Figure 2.1, it consists of stacked DDR-SDRAM dies, situated above a logic die through which the system accesses the memory [4, p. 1]. HBM is gradually being integrated into server processors, with the Intel® Xeon® Max Series [5] being one recent example. On these systems, different memory modes can be configured, 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].

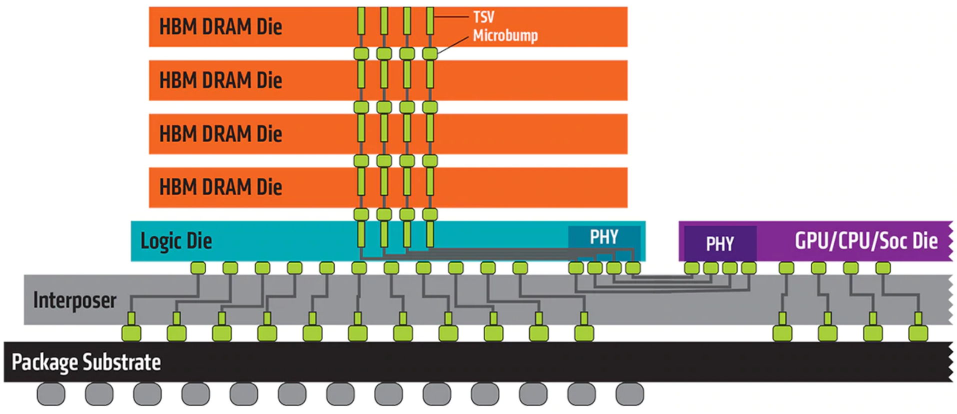


Figure 2.1: High Bandwidth Memory Design Layout. Shows that an HBM-module consists of stacked DRAM and a logic die. [3]

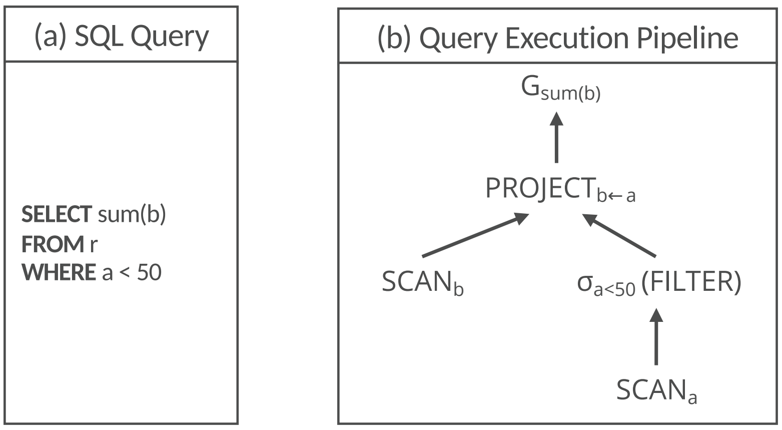


Figure 2.2: Illustration of a simple query in (a) and the corresponding pipeline in (b). [2, Fig. 1]

2.2 Query-driven Prefetching

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 2.2. From this plan, QdP determines columns in the database used in subsequent tasks. Once identified, the system proactively copies these columns into faster memory to accelerate subsequent operations by reducing the potential memory bottleneck. For the example (Figure 2.2), 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]

QdP processes tasks in parallel and in chunks, resulting in a high degree of concurrency. This increases demand for processing cycles and memory bandwidth. As prefetching takes place in parallel with query processing, it creates additional CPU load, potentially diminishing gains from the acceleration of subsequent steps through the cached data. Hence, our objective in this work is to offload the underlying copy operations to the DSA, reducing the CPU impact and thereby increasing the performance gains offered by prefetching. [2]

2.3 Intel Data Streaming Accelerator

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.

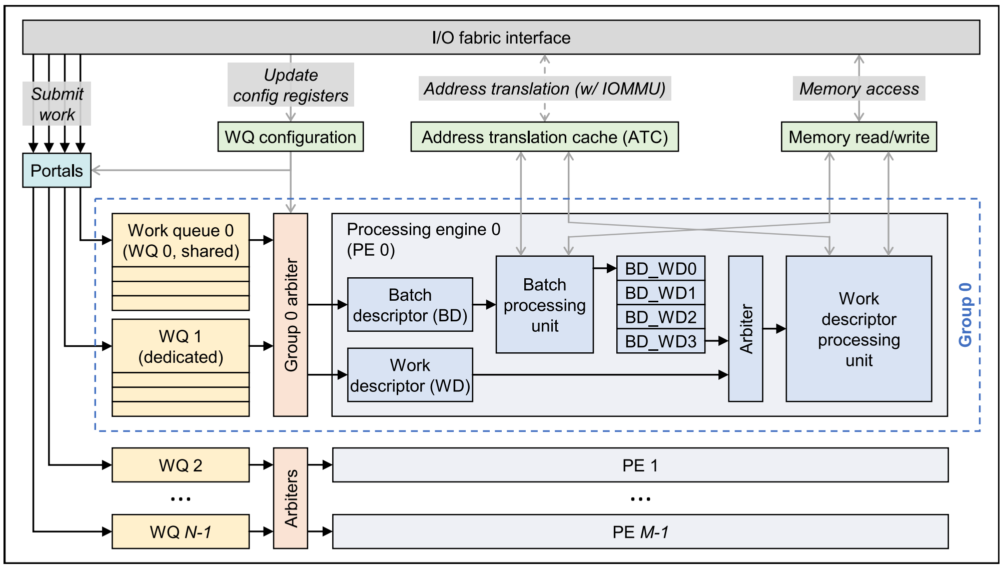


Figure 2.3: Intel Data Streaming Accelerator Internal Architecture. Shows the components that the chip is made up of, how they are connected and which outside components the DSA communicates with. [6, Fig. 1 (a)]

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 [7, Sec. 3]. Through this interface, the DSA is accessible and configurable as a PCIe device [7, Sec. 3.1]. 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: WQs provide the means to submit tasks to the device and are marked yellow in Figure 2.3. A Work Queue (WQ) is accessible through so-called portals, light blue in Figure 2.3, which are mapped memory regions to which a descriptor is written, facilitating task submission. 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. The method used to achieve this guarantee may result in higher submission cost [7, Sec. 3.3.1], compared to the Dedicated Work Queue (DWQ) to which a descriptor is submitted via a regular write [7, 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. Processing a task descriptor is straightforward, as all information required to complete the operation is contained within [7, Sec. 3.2]. For a batch, the DSA reads the batch descriptor, then fetches all task descriptors from memory and processes them [7, 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 [7, 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 2.3. Consequently, tasks from one WQ may be processed by one of multiple Engines and vice-versa, depending on the configuration. This flexibility is achieved through the Group Arbiter, represented by the orange block in Figure 2.3, which connects Engines and s according to the user-defined configuration.

2.3.1.2 Virtual Address Resolution

An important aspect of computer systems is the abstraction of physical memory addresses through virtual memory [9]. Therefore, the DSA must handle address translation because a process submitting a task will not know the physical location in memory of its data, causing the descriptor to contain virtual addresses. To resolve these to physical addresses, the Engine communicates with the Input/Output Memory Management Unit to perform this operation, as visible in the outward connections at the top of Figure 2.3. Knowledge about the submitting processes is required for this resolution. Therefore, each task descriptor has a field for the Process Address Space ID which is filled by the instruction used by SWQ submission [7, Sec. 3.3.1] or set statically after a process is attached to a DWQ [7, 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' [7, Sec. 3.7].

2.3.1.4 Ordering Guarantees

Ordering guarantees enable or restrict the safe submission of tasks to the DSA which depend on the completion of previous work on the accelerator. Guarantees are only given 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'. Consequently, when submitting a task mutating the value at address A, followed by one reading from A, the read may pass the write and therefore see the

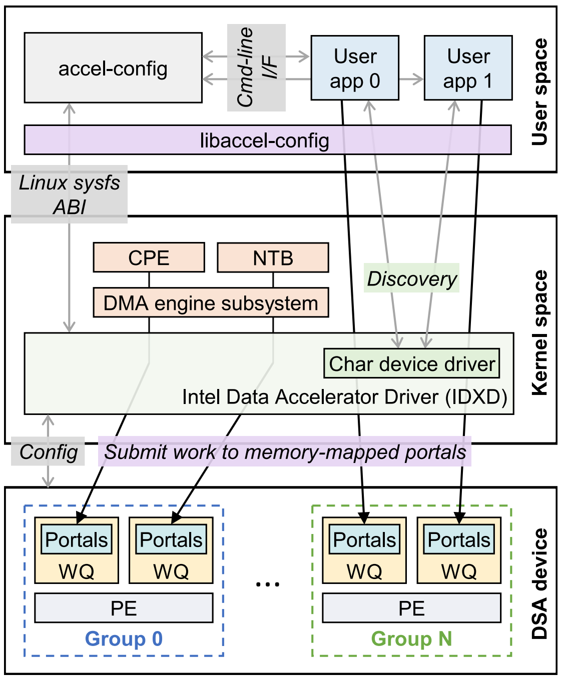


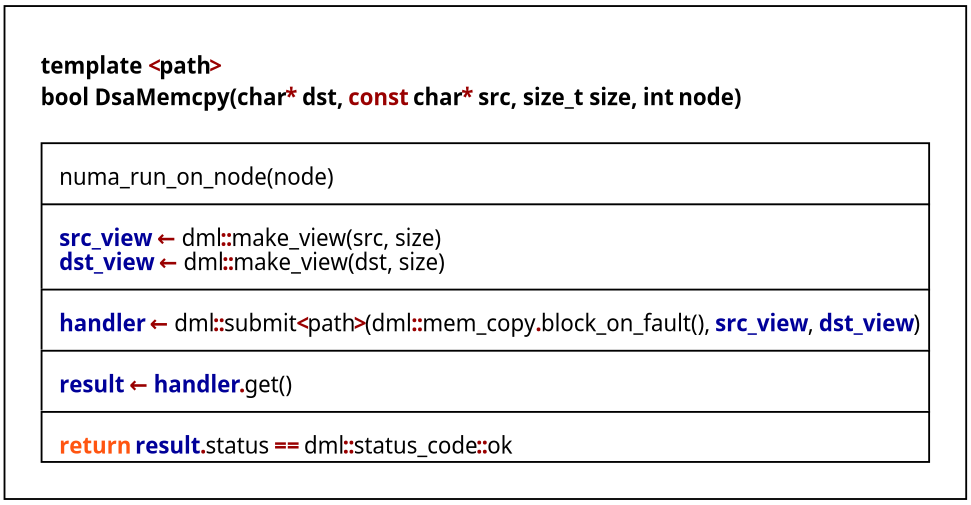
Figure 2.4: Intel Data Streaming Accelerator Software View. Illustrating the software stack and internal interactions from user applications, through the driver to the portal for work submission. [6, Fig. 1 (b)]

unmodified state of the data at A. Additional 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.

Operation ordering hazards 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. [7, Sec. 3.9]

2.3.2 Software View

At last, we will give an overview of the available software stack for interacting with the DSA. Driver support is limited to Linux, where a driver for the DSA has been available since Kernel version 5.10 [10, Sec. Installation]. As a result, we consider accessing the DSA under different operating systems unfeasible. To interact with the driver and perform configuration operations, Intel provides the accel-config user-space application and library [11]. This toolset offers a command-line interface and can read description files to configure the device (see Section 2.3.1 for information on this configuration), while also facilitating hardware discovery. The interaction is illustrated in the upper block labelled ‘User space’ in Figure 2.4, where it communicates with the kernel driver, depicted in light green and labelled ‘IDXD’ in Figure 2.4. Once successfully configured,



```
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 2.5: 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`. The template parameter `path` selects between hardware offloading (Intel DSA) or software execution (CPU).

each WQ is exposed as a character device through `mmap` of the associated portal [6, Sec. 3.3].

While a process could theoretically submit work to the DSA by manually preparing descriptors and submitting them via special instructions, this approach can be cumbersome. Hence, Intel Data Mover Library (Intel DML) exists to streamline this process. Despite some limitations, such as the lack of support for DWQ submission, this library offers an interface that manages the creation and submission of descriptors, as well as error handling and reporting. The high-level abstraction offered enables compatibility measures, allowing code developed for the DSA to also execute on machines without the required hardware [10, Sec. High-level C++ API, Advanced usage]. Section 2.4 provides an overview and example pseudocode for usage of Intel DML.

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.

In the function header of Figure 2.5 two differences from standard memcpy are notable. Firstly, there is the template parameter named `path`, and secondly, an additional parameter `node`. The `path` allows 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, favouring DSA where available [10, 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. This can either be achieved by pinning the current thread to the NUMA-Node (Node) that the device is located on, or by using optional parameters of `dml::submit` [10, Sec. High-level C++ API, NUMA support]. As evident from Figure 2.5, we chose the former option for this example, using `numa_run_on_node` to restrict the current thread to run on the Node chosen by `node`. Given that Figure 2.5 serves as an illustrative example, any potential side effects resulting from the modification of NUMA assignments through the execution of this pseudocode are disregarded.

Intel DML operates on data views, which we create from the given pointers to source and destination and size [10, Sec. High-level C++ API, Make view]. This is done using `dml::make_view(uint8_t* ptr, size_t size)`, visible in Figure 2.5, where these views are labelled `src_view` and `dst_view`. Following this preparation, we submit a single descriptor using the asynchronous operation from Intel DML. For submission, the function `dml::submit<path>` is used, 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. [10, Sec. High-level C++ API]

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 [12, 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 [13, p. 5]

2. Use a kernel with IDXD driver support, available from Linux 5.10 or later [10, Sec. Installation], and append the following to the kernel boot parameters in grub config: `intel_iommu=on,sm_on` [13, 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 [13, p. 5]

4. Install `accel-config` from source [11] or system package manager and inspect the detection of DSA devices through the driver using `accel-config list -i` [13, p. 6]

5. Create DSA configuration file for which we provide an example under `benchmarks/configuration-files/8n1d1e1w.conf` in the accompanying repository [14] that is also applied for the benchmarks. 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` [13, 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. As the performance of the DSA has been evaluated in great detail by Reese Kuper et al. in [6], we will perform only a limited amount of benchmarks with the purpose of determining behaviour in a multi-socket system, penalties from using Intel DML, and throughput for transfers from DDR-SDRAM to HBM.

3.1 Benchmarking Methodology

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 physical 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. [16]

As Intel DML does not have support for DWQs (see Section 2.4), we run benchmarks exclusively with access through SWQs. The application written for the benchmarks can be inspected in source form under the directory `benchmarks` in the thesis repository [14].

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

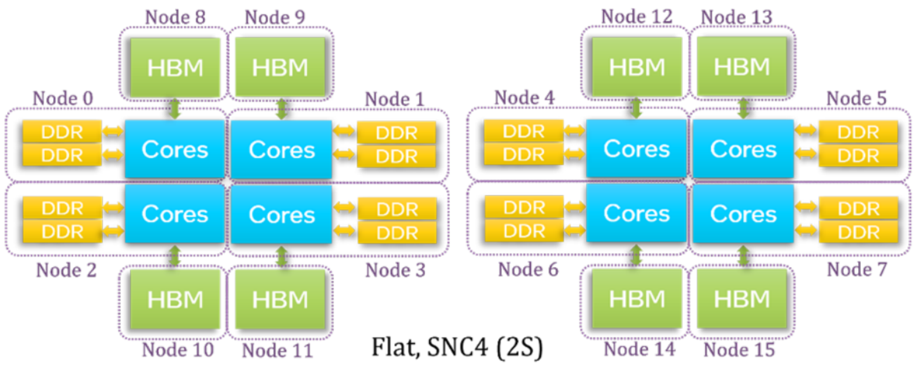
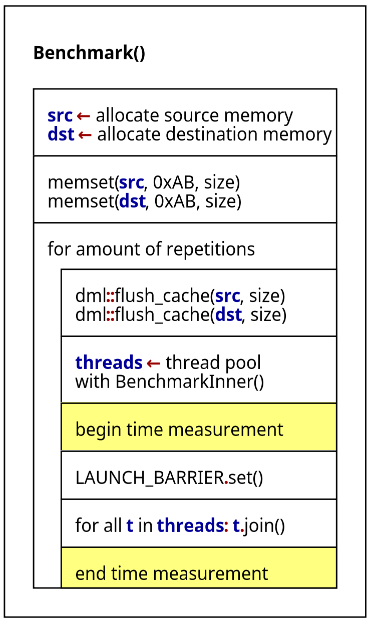


Figure 3.1: Xeon Max Layout [15, Fig. 14] for a 2-Socket System when configured with HBM-Flat. Showing separate Node IDs for manual HBM access and for Cores and DDR-SDRAM.



```
Benchmark()
    src ← allocate source memory
    dst ← allocate destination memory
    memset(src, 0xAB, size)
    memset(dst, 0xAB, size)
    for amount of repetitions
        dml::flush_cache(src, size)
        dml::flush_cache(dst, size)
        threads ← thread pool
        with BenchmarkInner()
        begin time measurement
        LAUNCH_BARRIER.set()
        for all t in threads: t.join()
        end time measurement
```

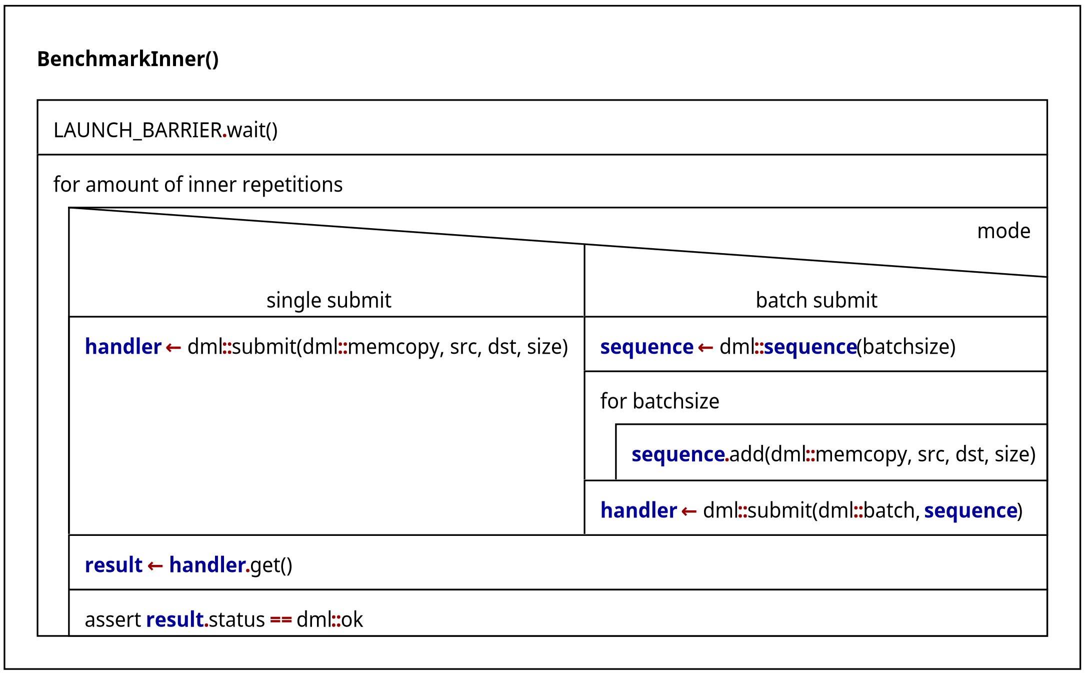
Figure 3.2: Outer Benchmark Procedure Pseudocode. Timing marked with yellow background. Showing preparation of memory locations, clearing of cache entries, timing points and synchronized benchmark launch.

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.

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 under 100 ms. This short execution window can have adverse effects on the timings. Therefore, we repeat the code of the inner loop for a configurable amount, virtually extending the duration of a single iteration for these cases. Figure 3.3 depicts this behaviour as the for-loop. The chosen internal repetition count is 10000 for transfers in the range of $1 - 8$ KiB, 1000 for 1 MiB, and one for larger instances.

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. This is shown by the accesses to 'LAUNCH_BARRIER' in both Figure 3.3 and 3.2. The behaviour in the inner function then differs depending on the submission method



```
BenchmarkInner()
LAUNCH_BARRIER.wait()
for amount of inner repetitions
    mode
        single submit:
            handler ← dml::submit(dml::memcopy, src, dst, size)
        batch submit:
            sequence ← dml::sequence(batchsize)
            for batchsize
                sequence.add(dml::memcopy, src, dst, size)
            handler ← dml::submit(dml::batch, sequence)
    result ← handler.get()
    assert result.status == dml::ok
```

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

selected which can be a single submission or a batch of given size. This selection is displayed 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 submission works unlike the former. A sequence with specified size is created which tasks are then added to. A prepared sequence is submitted to the engine similarly to a single descriptor.

3.2 Benchmarks

In this section, we will introduce three benchmarks, providing setup information and a brief overview of each. Subsequently, we will present plots illustrating the results, followed by a comprehensive analysis. Our approach involves formulating expectations and juxtaposing them with the observations derived 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.

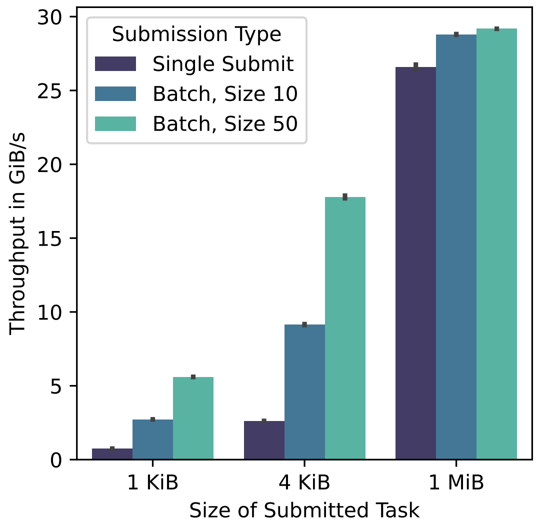


Figure 3.4: Throughput for different Submission Methods and Sizes. Performing a copy with source and destination being Node 0, executed by the DSA on Node 0. Observable is the submission cost which affects small transfer sizes differently.

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 it only once for multiple copies.

In Figure 3.4 we conclude that with transfers of 1 MiB and upwards, the cost of single submission drops. Even for transferring the largest datum in this benchmark, we still observe a speed delta between batching and single submission, pointing to substantial submission costs encountered for submissions to the SWQ and task preparation. For smaller transfers the performance varies greatly, with batch operations leading in throughput. Reese Kuper et al. noted that 'SWQ observes lower throughput between $1 - 8$ KB [transfer size]' [6, pp. 6]. We however measured a much higher point of equalization, pointing to additional delays introduced by programming the DSA through Intel DML. Another limitation is visible in our result, namely the inherent throughput limit per DSA chip of close to 30 GiB/s. This is caused by I/O fabric limitations [6, 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

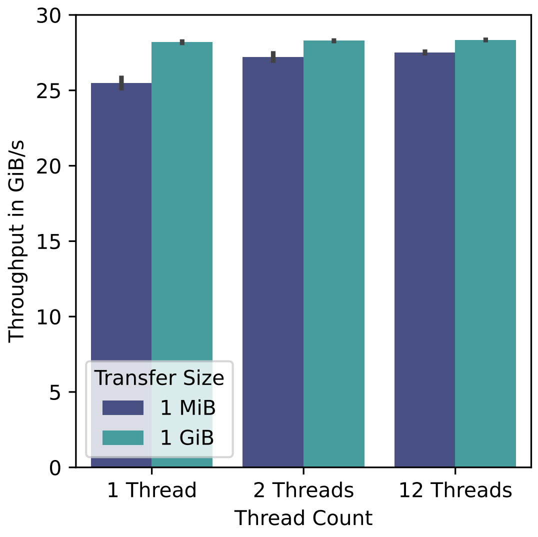


Figure 3.5: Throughput for different Thread Counts and Sizes. Multiple threads submit to the same Shared Work Queue. Performing a copy with source and destination being Node 0, executed by the DSA on Node 0.

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.

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.

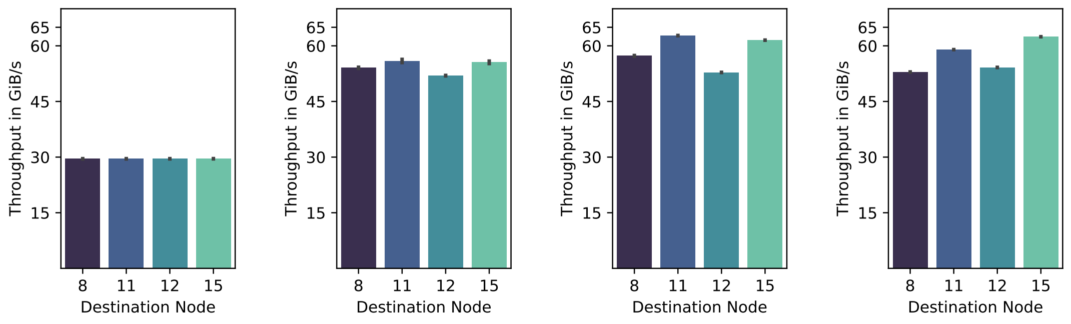
3.2.3 Data Movement from DDR-SDRAM to HBM

$$2\ DIMM \times \frac{4400\ MT}{s\ \times\ DIMM} = 8800\ MT/s \tag{3.1}$$

$$\frac{64b}{8b/B}\ /\ T = 8\ B/T \tag{3.2}$$

$$8800\ MT/s \times 8B/T = 70400 \times 10^6 B/s = 65.56\ GiB/s \tag{3.3}$$

Moving data from DDR-SDRAM to HBM is most relevant to the rest of this work, as it is the target application. With HBM offering higher bandwidth than the DDR-SDRAM of our system, we will be restricted by the available bandwidth of the source. To determine the upper limit achievable, we must calculate the available peak bandwidth. For each Node, the test system is configured with two DIMMs of DDR5-4800. The naming scheme contains the data rate in Megatransfers (MT) per second, however the



(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.

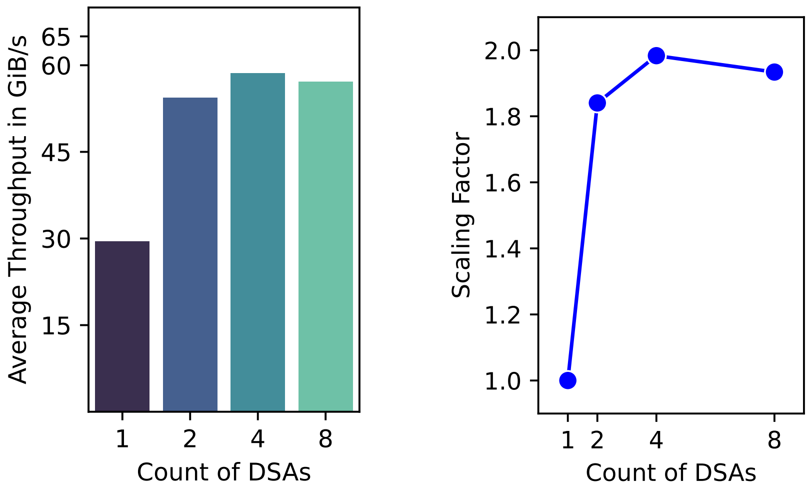
Figure 3.6: Copy from Node 0 to the destination Node specified on the x-axis. Shows peak throughput achievable with DSA for different load balancing techniques.

processor specification notes that for dual channel operation, the maximum supported speed drops to 4400 MT/s [16]. We calculate the transfers performed per second for one Node (1), followed by the bytes per transfer [17] in calculation (2), and at last combine these two for the theoretical peak bandwidth per Node on the system (3).

From the observed bandwidth limitation of a single DSA situated at about 30 GiB/s (see Section 3.2.1) and the available memory bandwidth of 65.56 GiB/s, we conclude that a copy task has to be split across multiple DSAs to achieve peak throughput. Different methods of splitting will be evaluated. Given that our system consists of multiple processors, data movement between sockets must also be evaluated, as additional bandwidth limitations may be encountered [18]. Beyond two DSA, marginal gains are to be expected, due to the throughput limitation of the available source memory.

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 utilize 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 GiB of data from Node 0 to the destination Node. We present data for Nodes 8, 11, 12, and 15. To understand the selection, see 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) Average Throughput for different amounts of participating DSA.

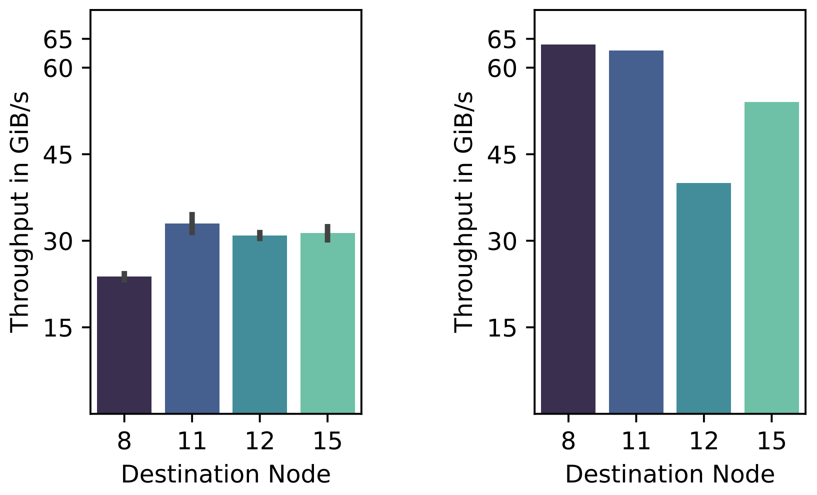
(b) Scaling Behaviour for different amounts of participating DSA. Displays factor of performance from utilizing one DSA.

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.

We begin by examining the common behaviour of load balancing techniques depicted in Figure 3.6. The real-world peak throughput of 64 GiB/s approaches the calculated available bandwidth. 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, Node 8 performs worse than copying to Node 11. As Node 8 serves as the HBM accessor for the data source Node, the data path is the shortest tested. The lower throughput suggests that the DSA may suffer from sharing parts of the path for reading and writing in this situation. 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, close to the calculated theoretical limit, 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



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

(b) Colleague's CPU peak throughput benchmark [19] 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.

using the off-socket DSAs. We will analyse this behaviour by comparing the different benchmarked configurations and summarize our findings on scalability.

When comparing the Brute-Force approach with Push-Pull in Figure 3.7b, average performance decreases by utilizing four times more resources over a longer duration. As shown in Figure 3.6b, using Brute-Force still leads to a slight increase in throughput for inter-socket operations, although far from scaling linearly. Therefore, we conclude that, although data movement across the interconnect incurs additional cost, no hard bandwidth limit is observable, reaching the same peak speed also observed for intra-socket with four DSAs. This might point to an architectural advantage, as we will encounter the expected speed reduction for copies crossing the socket boundary when executed on the CPU in Section 3.2.4.

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 processors' interconnect, however, as no hard limit is encountered, this is not a definitive answer. Utilizing off-socket DSAs proves disadvantageous in the average case, reinforcing the claim of communication overhead for crossing the socket boundary.

The choice of a load balancing method is not trivial. Consulting Figure 3.7a, the highest throughput is achieved by using four DSAs. At the same time, this causes high system utilization, making it unsuitable for situations where resources are to be distributed among multiple control flows. For this case, 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 previous Section (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 [19], from which we will present results as well. We compare expectations and results from the previous Section with the measurements.

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. In Figure 3.8b, peak throughput is achieved for intra-node operation, validating 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 observed in Section 3.2.3, can be seen in Figure 3.8. This points to an architectural anomaly which we could not explain with our knowledge or benchmarks. Further benchmarks were conducted for this, not yielding conclusive results, as the source and copy-thread location did not seem to affect the observed speed delta.

3.3 Analysis

In this section we summarize the conclusions from the performed benchmarks, outlining a utilization guideline.

- 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. Otherwise, offloading might prove more expensive than performing the copy on CPU.
- Section 3.2.2 assures that access from multiple threads does not negatively affect the performance when using a 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.
- Combining the result from Sections 3.2.3 and 3.2.1, we posit that for situations with smaller transfer sizes and a high amount of tasks, splitting a copy might prove disadvantageous. Due to incurring more delay from submission and overall throughput still remaining high without the split due to queue filling (see Section 3.2.2), the split might reduce overall effectiveness. To utilize the available resources effectively, distributing tasks across the available DSAs is still desirable. This finding led us to implement round-robin balancing in Section 5.2.

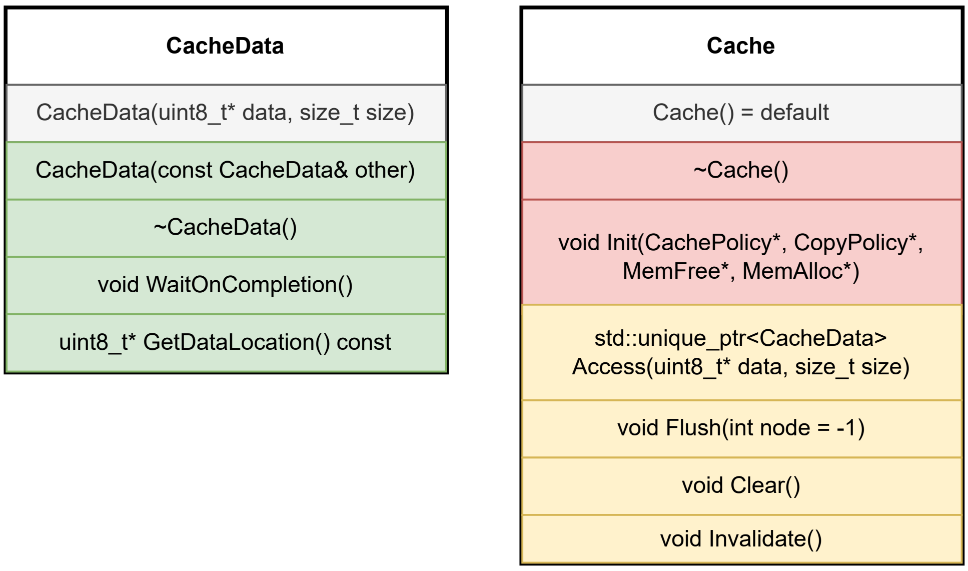
Once again, we refer to Figures 3.6 and 3.8, both representing the maximum throughput achieved with the utilization of either DSA for the former and CPU for the latter. Noticeably, the DSA does not seem to suffer from inter-socket overhead like the CPU. The DSA performs similar to the CPU for intra-node data movement, while outperforming it in inter-node scenarios. The latter, as mentioned in Section 3.2.3, might point to an architectural advantage of the DSA. The performance observed in the above benchmarks demonstrates potential for rapid data movement while simultaneously relieving the CPU of this task and thereby freeing capacity then available for computational tasks.

We encountered an anomaly on Node 12 for which we were unable to find an explanation. Since this behaviour is also observed on the CPU, pointing to a problem in the system architecture, identifying the root cause falls beyond the scope of this work. Despite being unable to account for all measurements, this chapter still offers valuable insights into the performance of the DSA, highlighting both its strengths and weaknesses. It provides data-driven guidance for a complex architecture, aiding in the determination of an optimal utilization strategy for the DSA.

4 Design

In this chapter, we design a class interface for a general-purpose cache. We will outline the requirements and elucidate the solutions employed to address them, culminating in the final architecture. Details pertaining to the implementation of this blueprint will be deferred to Chapter 5, where we delve into a selection of relevant aspects.

The target application of code contributed by this work is to accelerate Query-driven Prefetching by offloading copy operations to the DSA. Prefetching is inherently related with cache functionality. Given that an application providing the latter offers a broader scope of utility beyond QdP, we opted to implement an offloading `Cache`.



CacheData
CacheData(uint8_t* data, size_t size)
CacheData(const CacheData& other)
~CacheData()
void WaitOnCompletion()
uint8_t* GetDataLocation() const
Cache
Cache() = default
~Cache()
void Init(CachePolicy*, CopyPolicy*, MemFree*, MemAlloc*)
std::unique_ptr<CacheData> Access(uint8_t* data, size_t size)
void Flush(int node = -1)
void Clear()
void Invalidate()

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.

4.1 Interface

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. Operation (3) is required when the data that is cached may also be modified, necessitating synchronization between cache and source memory. Due to various setups and use cases for the 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 unused entries, when memory pressure during access of a new memory block demands it.

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. Operations (1) and (2) are provided by one single function, which we call Cache::Access. This function receives a data pointer and size as parameters. Its behaviour depends on the state of the cache containing an entry for the requested block, only submitting a caching operation if the pointer received is not yet cached and returning the cache entry in both cases. Operations Cache::Access and Cache::Invalidate, and additional methods, deemed useful but not covered by the behavioural requirements, of the Cache-class can be viewed in Figure 4.1.

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 and allowing work to continue in parallel. To facilitate this process, the Cache::Access method returns an instance of the class CacheData. Figure 4.1 also documents the public interface for CacheData in the left block. 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. A call to CacheData::WaitOnCompletion is required should the user desire to use the cache, as only through it, the cache pointer is updated.

4.1.1 Policy Functions

In the introduction of this chapter, we mentioned placing cache placement and selecting copy-participating DSAs in the responsibility of the user. As we will find out in Section 5.2, allocating memory inside the cache is not feasible due to possible delays encountered. Therefore, the user is also required to provide functionality for dynamic memory management to the Cache. The former is realized by what we will call 'Policy Functions', which are function pointers passed on initialization, as visible in Cache::Init in Figure 4.1. We use the same methodology for the latter, requiring functions performing dynamic memory management. As the choice of cache placement and copy policy is user-defined, one possibility will be discussed for the implementation in Chapter 5, while we detail their required behaviour here.

The policy functions receive arguments deemed sensible for determining placement and participation selection. Both are informed of the source , the requesting caching, and the data size. The cache placement policy then returns a -ID on which the data is to be cached, while the copy policy will provide the cache with a list of -IDs, detailing which DSAs should participate in the operation.

For memory management, two functions are required, providing allocation (`malloc`) and deallocation (`free`) capabilities to the `Cache`. Following the naming scheme, these two functions must adhere to the thread safety guarantees and behaviour set forth by the C++ standard. Most notably, `malloc` must never return the same address for subsequent or concurrent calls.

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 entry reuse, the existing `CacheData` will be extended in scope to handle multiple consumers. Copies of it can be created, and 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. The `Cache` must then ensure that, on concurrent access to the same resource, only one thread creates `CacheData` while the others are provided with a copy.

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 Weak Behaviour and Page Fault Handling

During our testing phase, we discovered that Intel DML does not support interrupt-based completion signaling, as discussed in Section 2.3.1.3. Instead, it resorts to busy-waiting, which consumes CPU cycles, thereby impacting concurrent operations. To mitigate this issue, we extended the functionality of both `Cache` and `CacheData`, providing weak versions of `Cache::Access` and `CacheData::WaitOnCompletion`. The weak wait function only checks for operation completion once and then returns, irrespective of the completion state, thereby relaxing the guarantee that the cache location will be valid after the call. Hence, the user must verify validity after the wait if they choose to utilize this option. Similarly, weak access merely returns a pre-existing instance of `CacheData`, bypassing work submission when no entry is present for the requested

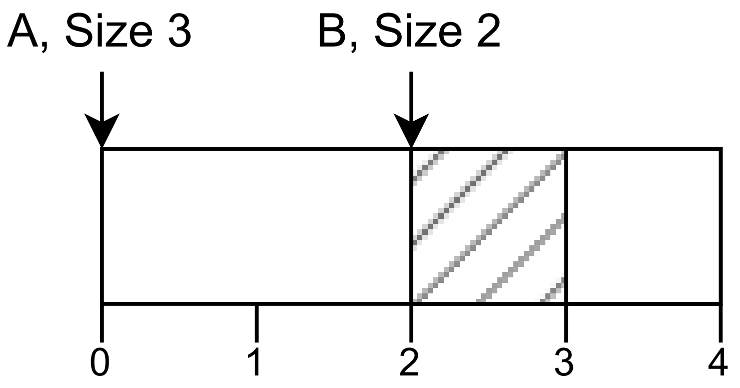


Figure 4.2: Public Interface of `CacheData` and `Cache` Classes. Colour coding for thread safety. Grey denotes impossibility for threaded access. Green indicates full safety guarantees only relying on atomics to achieve this. Yellow may use locking but is still safe for use. Red must be called from a single threaded context.

memory block. These features prove beneficial in latency-sensitive scenarios where the overhead from cache operations and waiting on completion is undesirable. As the weak functions constitute optional behaviour, not integral to the `Caches'` operation, Figure 4.1 does not cover them.

Additionally, while optimizing for access latency, we encountered delays caused by page fault handling on the DSA. These not only affect the current task but also impede the progress of other tasks on the DSA, by blocking. Consequently, the `Cache` defaults to not handle page faults.

To configure runtime behaviour for page fault handling and access type, we introduced a flag system to both `Cache` and `CacheData`, with the latter inheriting any flags set in the former upon creation. This design allows for global settings, such as opting for weak waits or enabling page fault handling. Weak waits can also be selected in specific situations by setting the flag on the `CacheData` instance. For `Cache::Access`, the flag must be set for each function call, defaulting to strong access, as exclusively using weak access would result in no cache utilization.

4.2 Usage Restrictions

In the context of this work, the cache primarily manages static data, leading to two restrictions placed on the invalidation operation, allowing significant reductions in design complexity. Firstly, due to the cache's design, accessing overlapping areas will lead to undefined behaviour during the invalidation of any one of them. Only entries with equivalent source pointers will be invalidated, while other entries with differing source pointers, still covering the now-invalidated region due to their size, will remain unaffected. Consequently, the cache may or may not continue to contain invalid elements. This scenario is depicted in Figure 4.2, where A points to a memory region, which due to its size, overlaps with the region pointed to by B. Secondly, invalidation is a manual process, necessitating the programmer to recall which data points are currently cached and to invalidate them upon modification. In this scenario, no ordering guarantees are provided, potentially resulting in threads still holding pointers to now-outdated entries and continuing their progress with this data.

Additionally, the cache inherits some restrictions of the DSA. Due to the asynchronous operation and internal workings, multiple operation ordering hazards (Section 2.3.1.4) may arise, either internally within one DSA, or externally when coming in conflict with other DSAs. However, these hazards do not impact the user of `Cache`, as the design implicitly prevents hazardous situations. Specifically, `CacheData::WaitOnCompletion` ensures that memory regions written to by the DSA are only accessible after operations have completed. This guards against both intra- and inter-accelerator ordering hazards. Memory regions that a DSA may access are internally allocated in the `Cache` and become retrievable only after operation completion is signalled, thus preventing submissions for the same destination. Additionally, multiple threads submitting tasks for the same source location does not pose a problem, as the access is read-only, and is also prevented by `Cache::Access` only performing work submission on one thread.

Despite accounting for the hazards in operation ordering, one potential situation may still lead to undefined behaviour. Since the DSA operates asynchronously, modifying data for a region present in the cache before ensuring that all caching operations have completed, through a call to `CacheData::WaitOnCompletion`, will result in an undefined state for the cached region. Therefore, it is crucial to explicitly wait for caching operations to complete before modification of the source memory to avoid such scenarios.

Conclusively, the `Cache` should not be used with overlapping buffers. Caching mutable data presents design challenges which could be solved by implementing a specific templated data type, which we will call `CacheMutWrapper`. This type internally stores its data in a struct which can then be cached and is tagged with a timestamp. `CacheMutWrapper` then ensures that all Mutating Method also invalidate the cache, perform waiting on operation completion to avoid modification while the DSA accesses the cached region, and update the timestamp. This stamp is then used by further processing steps to verify that a value presented to them by a thread was calculated using the latest version of the data contained in `CacheMutWrapper`, guarding against the lack of ordering guarantees for invalidation, as mentioned by the first paragraph of this section.

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. First, we delve into the usage of locking and atomics to achieve thread safety and then apply the cache to Query-driven Prefetching, thereby detailing the policies mentioned in Section 4.1.1 and presenting solutions for the challenges encountered.

5.1 Synchronization for Cache and CacheData

The usage of locking and atomics to achieve safe concurrent access 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 focus on the synchronization techniques employed.

Throughout the following sections we will use the term ‘handler’, which was coined by Intel DML, referring to an object associated with an operation on the accelerator, exposing the completion state of the asynchronously executed task. Use of a handler is also displayed in the `memcpy`-function for the DSA as shown in Figure 2.5. 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.

5.1.1 Cache: Locking for Access to State

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 design decision 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`. Both flushing and clear operations necessitate 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 when adding the newly created entry to cache.

A map was chosen as the data structure to represent the current cache state with the key being the memory address of the entry, holding a `CacheData` instance as value. Thereby we achieve the promise of keeping entries cached whenever possible. With the `Cache` keeping an internal reference, the last reference to `CacheData` will only be destroyed upon removal of the cache state. This removal can either be triggered implicitly by flushing when encountering Memory Pressure during a cache access, or explicitly by calls to `Cache::Flush` or `Cache::Clear`.

With the caching policy being 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 also reduces the potential for lock contention by separately locking each Node's state instead of utilizing a global lock. Even with this optimization, scenarios where the `Cache` is frequently tasked with flushing and re-caching by multiple threads from the same node, will lead to threads contending for unique access to the state.

5.1.2 CacheData: Shared Reference

The choice made in 4.1.2 necessitates thread-safe shared access to one instance of `CacheData`. The C++ standard library provides `std::shared_ptr<T>`, a reference-counted pointer that is thread-safe for the required operations [20], making it a suitable candidate for this task. Although an implementation using it was explored, it presented its own set of challenges. Using `std::shared_ptr<T>` additionally introduces uncertainty, relying on the implementation to be performant. The standard does not specify whether a lock-free algorithm is to be used, and [21] suggests abysmal performance for some implementations. 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.

5.1.3 CacheData: Fair and Threadsafe WaitOnCompletion

As Section 4.1.2 details, we intend to share a cache entry between multiple threads, necessitating synchronization between multiple instances of `CacheData` for waiting on operation completion and determining the cache location. While the latter is easily achieved by making the cache pointer a shared atomic value, the former proved to be challenging. Therefore, we will iteratively develop our implementation for `CacheData::WaitOnCompletion` in this Section. We present the challenges encountered and describe how these were solved to achieve the fairness and thread safety we desire.

We assume the handlers of Intel DML to be unsafe for access from multiple threads, as no guarantees were found in the documentation. To achieve the safety for `CacheData::WaitOnCompletion`, outlined in Section 4.1.2, threads need to coordinate on a master thread which performs the actual waiting, while the others wait on the master.

Upon call to `Cache::Access`, coordination must take place to only add one instance of `CacheData` to the cache state and, most importantly, submit to the DSA only once. This behaviour was elected in Section 4.1.2 to reduce the load placed on the accelerator by preventing duplicate submission. To solve this, `Cache::Access` will add the instance to the cache state under unique lock (see Section 5.1.1 above) and only then, when the current thread modified the cache state, submission will take place. This presents a data

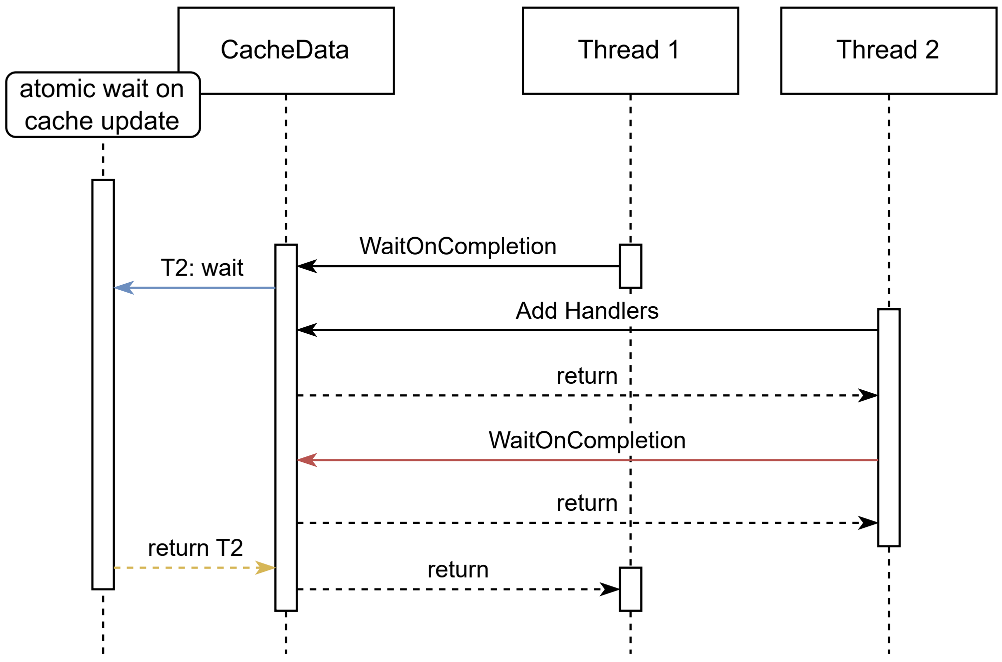
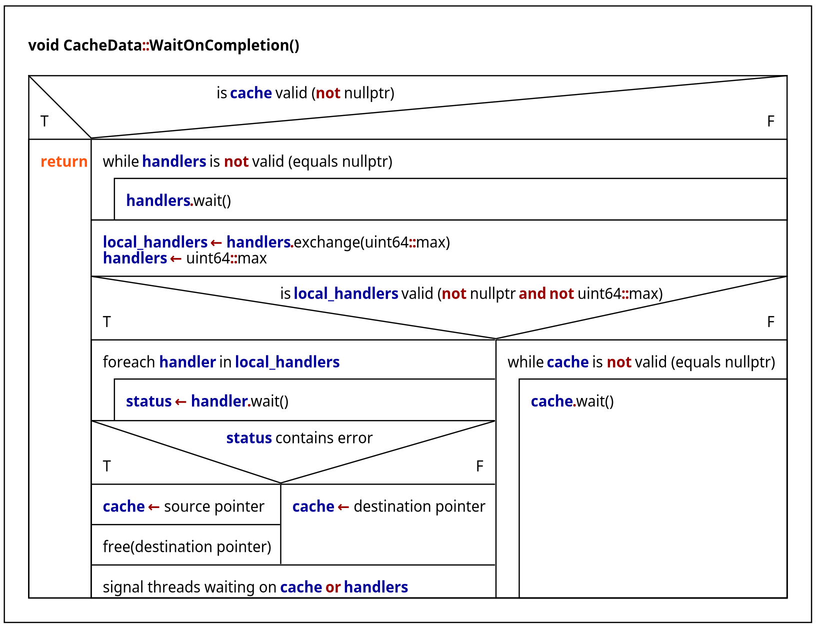


Figure 5.1: Sequence for Blocking Scenario. Observable in first draft implementation. Scenario where T_1 performed first access to a datum followed T_2. Then T_1 holds the handlers exclusively, leading to T_2 having to wait for T_1 to perform the work submission and waiting.

race, with one thread successfully adding a new entry to the cache. This thread is then chosen to perform the submission. Through this two-stepped process, the state may contain `CacheData` without a valid cache pointer and no handlers available to wait on. As the following paragraph and sections demonstrate, this resulted in implementation challenges.

In the first implementation, a thread would check if the handlers are available and wait 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 two threads T_1 and T_2 wish to access the same resource. 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 obtains access to the incomplete `CacheData` on which it calls `CacheData::WaitOnCompletion`, causing it to see `nullptr` for the cache pointer and handlers. Therefore, T_2 waits on the cache pointer becoming valid (marked blue lines in Figure 5.1). Then T_1 submits the work and sets the handlers (marked red lines in Figure 5.1), while T_2 continues to wait on the cache pointer. Therefore, only T_1 can trigger the waiting on the handlers and is therefore capable of keeping T_2 from progressing. This is undesirable as it can lead to deadlocking if T_1 does not wait and at the very least may lead to unnecessary delay for T_2.



```
void CacheData::WaitOnCompletion()
is cache valid (not nullptr)
T:
    return
F:
    while handlers is not valid (equals nullptr)
        handlers.wait()
    local_handlers ← handlers.exchange(uint64::max)
    handlers ← uint64::max
    is local_handlers valid (not nullptr and not uint64::max)
    T:
        foreach handler in local_handlers
            status ← handler.wait()
        status contains error
        T:
            cache ← source pointer
            free(destination pointer)
        F:
            cache ← destination pointer
        signal threads waiting on cache or handlers
    F:
        while cache is not valid (equals nullptr)
            cache.wait()
```

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

To resolve this, a more intricate implementation is required, for which Figure 5.2 shows pseudocode. When waiting, the threads now immediately check whether the cache pointer contains a valid value and return if it does. We will use the same example as before to illustrate the second part of the waiting procedure. Thread T_2 arrives in this latter section as the cache was invalid at the point in time when waiting was called for. They now wait on the handlers-pointer to change. When T_1 supplies the handlers after submitting work, it also uses `std::atomic<T>::notify_one` to wake at least one thread waiting on value change of the handlers-pointer. Through this, the exclusion that was observable in the first implementation is already avoided. The handlers will be atomically set to a valid pointer and a thread may pass the wait. Following this, the handlers-pointer is atomically exchanged, invalidating it and assigning the previous valid value to a local variable. Each thread checks whether it has received a valid pointer to the handlers from the exchange. If it has then the atomic operation guarantees that is now in sole possession of the pointer and the designated master-thread. The master is tasked with waiting, while other threads will now regress and call `CacheData::WaitOnCompletion` again, leading to a wait on the master thread setting the cache to a valid value.

5.1.4 CacheData: Edge Cases and Deadlocks

With the outlines of a fair implementation of `CacheData::WaitOnCompletion` drawn, we will now move our focus to the safety of `CacheData`. Specifically the following Sections will discuss possible deadlocks and their resolution.

5.1.4.1 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, which can be seen in Figure 5.2. When the handlers will be set in the future by the thread calling `Cache::Access` first, progress is guaranteed.

5.1.4.2 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, due to the wait on handlers as shown in Figure 5.2. To circumvent this, 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.

5.1.4.3 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, resulting in both the handlers and the cache pointer being `nullptr`. This results in an invalid state, like the one discussed in Section 5.1.4.1. 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. The employed solution can be seen in Figure 5.2, where after processing the handlers we check for errors in their results and upon encountering an error, the cache pointer is set to the data source. Other threads could have accumulated, waiting for the cache to become valid, as the handlers were already invalidated before. By setting the cache to a valid value, these threads are released and any subsequent calls to `CacheData::WaitOnCompletion` will immediately return. Therefore, the `Cache` does not guarantee that data will actually be cached, however, as Chapter 4 does not make such a promise, we decided on implementing this edge case handling.

5.1.4.4 Locally Invalid State due to Race Condition

The guarantee of `std::atomic<T>::wait` to only wake up when the value has changed [22] 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 which would allow resetting the pointer back to `nullptr` 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 solution for this is to not exchange the handlers-pointer with `nullptr` but with a second invalid pointer. We must determine a value for use in the exchange and therefore introduce a new attribute to `Cache`. The address is shared with `CacheData` upon creation, where it is then used in `CacheData::WaitOnCompletion`, exchanging the handlers with it, instead of `nullptr`. 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 by entering the false-branch of the validity check for 'local_handlers', as shown in Figure 5.2.

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. This causes less delay from 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 signalling (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 and added the possibility for weak access in Section 4.1.4. 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 in Section 4.1.1. Consequently, the benchmark can pre-allocate the required memory blocks, trigger page mapping, and subsequently pass these regions to the `Cache`. When these memory management functions guarantee that pages will be mapped or access latency is more important than using the cache, the user can then elect to forego enabling page fault handling on the DSA, as enabled by Section 4.1.4.

6 Evaluation

In this chapter, we establish anticipated outcomes for incorporating the developed `Cache` into Query-driven Prefetching, followed by a comprehensive assessment of the achieved results. The specifics of the benchmark are elaborated upon in Section 5.2. We conclude with a discussion of the `Cache` and the results observed.

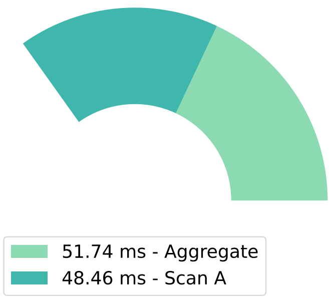
6.1 Benchmarked Task

The benchmark involves the execution of a simple query, which, as depicted in the execution plan in Figure 2.2, is divided into multiple processing stages. We will use the following notations for the three tasks performed in executing the query: (1) $SCAN_a$, responsible for scanning and subsequently filtering column `a`; (2) $SCAN_b$, tasked with prefetching column `b`; (3) $AGGREGATE$, the projection and final summation step. The column size utilized is set at 4 GiB. The workload is distributed across multiple groups, with each group spawning threads for every step. To ensure equitable comparison, each tested configuration employs 64 threads for the initial stage ($SCAN_a$ and $SCAN_b$) and 32 subsequently ($AGGREGATE$), while being constrained to execute on Node 0 through pinning. For configurations without prefetching, $SCAN_b$ is omitted. We measure total and per-task duration and cache hit ratio for prefetching over five iterations, forming the average. This is preceded by five warm-up runs, for which no timings are recorded.

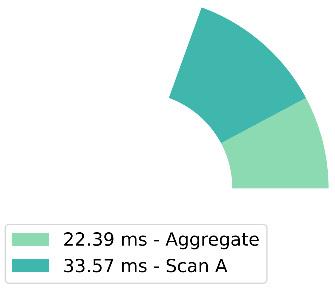
The tasks $SCAN_a$ and $SCAN_b$ execute concurrently, completing their tasks before signalling $AGGREGATE$ for finalization. In a bid to enhance the cache hit rate, we opted to relax this constraint, allowing $SCAN_b$ to operate independently, while only synchronizing $SCAN_a$ with $AGGREGATE$. This burst-submission could cause the DSAs work queue to overflow, leading us to increase the size of the processed chunks for the benchmarks utilizing QdP to reduce the amount of cached blocks.

6.2 Expectations

The simple query presents a challenging scenario for the `Cache`. The execution time for the filter operation applied to column `a` is expected to be brief. Consequently, the `Cache` has limited time for prefetching, which may exacerbate delays caused by processing overhead in the `Cache` or during accelerator offload. Furthermore, it can be assumed that $SCAN_a$ is memory-bound by itself. Since the prefetching of `b` in $SCAN_b$ and the loading and subsequent filtering of `a` occur concurrently, caching directly reduces the memory bandwidth available to $SCAN_a$ when both columns are located on the same Node.



(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.

Configuration	Raw Time
DDR-SDRAM (Baseline)	131.18 ms
HBM (Upper Limit)	93.09 ms

Table 6.1: Table showing raw timing for QdP on DDR-SDRAM and HBM. Result for DDR-SDRAM serves as baseline. HBM presents the upper boundary achievable, as it simulates prefetching without processing overhead and delay.

6.3 Observations

In this section, we will present our findings from integrating the `Cache` developed in Chapters 4 and 5 into QdP. We commence by presenting results obtained without prefetching, which serve as a reference for evaluating the effectiveness of our `Cache`. For all results presented, the amount of threads per processing stage and the amount of groups influences performance [2], which however is out of scope for this work. Therefore, results shown are for the best configurations measured.

The plots for detailed timing are normalized so that the longest running configuration fills the half-circle. As waiting times at the barriers, which can vary by workload, are not displayed here, the graphs do not fully represent the total execution time. Additionally, the total runtime also encompasses some overhead that the per-task timings do not cover. Therefore, a discrepancy between the raw runtime values from the Tables and the Figures may be observed.

6.3.1 Benchmarks without Prefetching

We benchmarked two methods to establish a baseline and an upper limit as reference points. In the former, all columns are located in DDR-SDRAM. The latter method

Configuration	Speedup	Cache Hitrate	Raw Time
DDR-SDRAM (Baseline)	x1.00	—	131.18 ms
HBM (Upper Limit)	x1.41	—	93.09 ms
Prefetching	x0.82	89.38 %	159.72 ms
Prefetching, Distributed Columns	x1.23	93.20 %	106.52 ms

Table 6.2: 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.

simulates perfect prefetching without delay and overhead by placing column `b` in HBM during benchmark initialization.

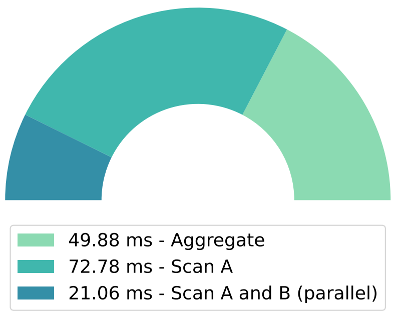
From Table 6.1, it is evident that accessing column `b` through HBM results in an increase in processing speed. To gain a better understanding of how the increased bandwidth of HBM accelerates the query, we will delve deeper into the time spent in the different stages of the query execution plan.

Due to the higher bandwidth provided by accessing column `b` through HBM in $AGGREGATE$, processing time is shortened, as demonstrated by comparing Figure 6.1b to the baseline depicted in Figure 6.1a. Consequently, more threads can be assigned to $SCAN_a$, with $AGGREGATE$ requiring less resources. This explains why the HBM-results not only show faster processing times than DDR-SDRAM for $AGGREGATE$ but also for $SCAN_a$.

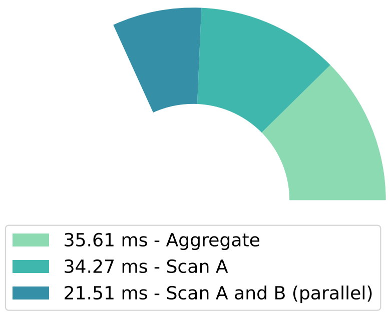
6.3.2 Benchmarks using Prefetching

To address the challenges posed by sharing memory bandwidth between both $SCAN$-operations, we will conduct the prefetching benchmarking in two configurations. Firstly, both columns `a` and `b` will be situated on the same Node. We anticipate demonstrating the memory bottleneck in this scenario, through increased execution time of $SCAN_a$. Secondly, we will distribute the columns across two Nodes, both still utilizing DDR-SDRAM. In this configuration, the memory bottleneck is alleviated, leading us to anticipate better performance compared to the former setup.

We now examine Table 6.2, where a slowdown is shown for prefetching. This drop-off below our baseline when utilizing the `Cache` may be surprising at first glance. However, it becomes reasonable when we consider that in this scenario, the DSAs executing the caching tasks compete for bandwidth with the threads processing $SCAN_a$, while also adding additional overhead from the `Cache` and work submission. The second measured configuration for QdP is shown as 'Prefetching, Distributed Columns' in Table 6.2. For this method, distributing the columns across different Nodes results in a noticeable performance increase compared to our baseline, although not reaching the upper boundary set by simulating perfect prefetching ('HBM (Upper Limit)' in Table



(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.

6.2). This confirms our assumption that $SCAN_a$ itself is bandwidth-bound, as without this contention, we observe an increase in cache hit rate and decrease in processing time. We will now examine the performance in more detail with per-task timings.

In Figure 6.2a, the competition for bandwidth between $SCAN_a$ and $SCAN_b$ is evident, with $SCAN_a$ showing significantly longer execution times. $SCAN_b$ is nearly unaffected, as it offloads memory access to the DSA through the `Cache`, thereby not showing extended runtime from the throughput bottleneck. This prolonged duration of execution in $SCAN_a$ leads to extended overlaps between groups still processing the scan and those engaged in $AGGREGATE$. Consequently, despite the relatively high cache hit rate (see Table 6.2), minimal speed-up is observed for $AGGREGATE$ compared to the baseline depicted in Figure 6.1a. The extended runtime can be attributed to the prolonged duration of $SCAN_a$.

Regarding the benchmark depicted in Figure 6.2a, where we distributed columns `a` and `b` across two nodes, the parallel execution of prefetching tasks on DSA does not directly impede the bandwidth available to $SCAN_a$. However, there is a discernible overhead associated with cache utilization, as evident in the time spent in $SCAN_b$. Consequently, both $SCAN_a$ and $AGGREGATE$ operations experience slightly longer execution times than the theoretical peak our upper-limit in Figure 6.1b exhibits.

6.4 Discussion

In Section 6.2, we anticipated that the simple query would pose a challenging case for prefetching. This expectation proved to be accurate, highlighting that improper data distribution can lead to adverse effects on performance when utilizing the `Cache`. Thus, we consider the chosen scenario to be well-suited, as it showcases both performance gains and losses, underscoring the importance of optimizing parameters and scenarios to achieve positive outcomes.

The necessity to distribute data across Nodes is seen as practical, given that developers commonly apply this optimization to leverage the available memory bandwidth of Non-Uniform Memory Architectures. Consequently, the `Cache` has demonstrated its effectiveness by achieving a respectable speed-up positioned directly between the baseline and the theoretical upper limit (see Table 6.2).

As stated in Chapter 4, the decision to design and implement a cache instead of focusing solely on prefetching was made to enhance the usefulness of this work's contribution. While our tests were conducted on a system with HBM, other advancements in main memory technologies, such as NVRAM, were not considered. Despite the methods of the `Cache`-class being named with usage as a cache in mind, its utility extends beyond this scope, providing flexibility through the policy functions, described in Section 4.1.1. Potential applications include replication to NVRAM for data loss prevention, or restoring from NVRAM for faster processing. Therefore, we consider the increase in design complexity to be a worthwhile trade-off, providing a significant contribution to the field of heterogeneous memory systems.

7 Conclusion And Outlook

In this work, our aim was to analyse the architecture and performance of the Intel Data Streaming Accelerator and integrate it into Query-driven Prefetching. We characterized the hardware and software architecture of the DSA in Section 2.3 and provided an overview of the available programming interface, Intel Data Mover Library, in Section 2.4. Our benchmarks were tailored to the planned application and included evaluations such as copy performance from DDR-SDRAM to HBM (Section 3.2.3), the cost of multithreaded work submission (Section 3.2.2), and an analysis of different submission methods and sizes (Section 3.2.1). Notably, we observed an anomaly in inter-socket copy speeds and found that the scaling of throughput was distinctly below linear (see Figure 3.7b). Although not all observations were explainable, the results provided important insights into the behaviour of the DSA and its potential application in multi-socket systems and HBM, complementing existing analyses [6].

Upon applying the cache developed in Chapters 4 and 5 to QdP, we encountered challenges related to available memory bandwidth and the lack of feature support in the API used to interact with the DSA. While the `Cache` represents a substantial contribution to the field, its applicability is constrained to data that is infrequently mutated. Although support exists for entry invalidation, it is rather rudimentary, requiring manual invalidation and the developer to keep track of cached blocks and ensure they are not overlapping (see Section 4.2). To address this, a custom container data type could be developed to automatically trigger invalidation through the cache upon modification and adding age tags to the data, which consumer threads can pass on. This tagging can then be used to verify that a threads work was performed on current data.

In Section 6.3, we observed adverse effects when prefetching with the cache during the parallel execution of memory-bound operations. This necessitated data distribution across multiple NUMA-Nodes to circumvent bandwidth competition caused by parallel caching operations. Despite this limitation, we do not consider it a major fault of the `Cache`, as existing applications designed for NUMA systems are likely already optimized in this regard.

As highlighted in Sections 2.4 and 5.2, the Application Programming Interface (API) utilized to interact with the DSA currently lacks support for interrupt-based completion waiting and the use of Dedicated Work Queue. Future development efforts may focus on direct DSA access, bypassing the Intel Data Mover Library, to leverage features of the DSA not implemented in the library. Particularly, interrupt-based waiting would significantly enhance the usability of the `Cache`, currently only supporting busy-waiting. This lead us to extend the design by implement weak-waiting in Section 5.2, favouring cache misses instead of wasting resources during the wait. Additionally, access through a

Dedicated Work Queue has the potential to reduce submission cost and thereby increase the `Caches'` effectiveness.

Although the preceding paragraphs and the results in Chapter 6 might suggest that the `Cache` requires extensive refinement for production applications, we argue the opposite. Under favourable conditions we observed significant speed-up using the `Cache` for prefetching to High Bandwidth Memory, accelerating database queries. Given that these conditions align with those typically found in NUMA-optimized applications, such a prerequisite is not unrealistic to expect. The utility of the `Cache` is not limited to prefetching alone; it offers a solution for replicating data to or from NVRAM and might prove applicable to other use cases. Additional benchmarks on more complex queries for QdP and a comparison between prefetching to HBM and 'HBM Cache Mode' (see Section 2.1) could yield deeper insights into the `Caches'` performance.

In conclusion, the developed library together with our exploration of architecture and performance of the DSA fulfil the stated goal of this work. We have achieved performance gains through offloading data movement for QdP, thereby demonstrating the DSAs potential to facilitate the exploitation of the properties offered by the various storage tiers in heterogeneous memory systems.

Glossary

API

Application Programming Interface: Definition of the interface provided by an application, enabling interaction between software components or systems.

DDR-SDRAM

Double Data Rate Synchronous Dynamic Random Access Memory: Main memory technology found in common computer systems.

DSA

Intel Data Streaming Accelerator: A component of modern Intel server processors, capable of executing common data operations asynchronously and thereby offloading them from the CPU.

DWQ

Dedicated Work Queue: A type of Work Queue only usable by one process, and therefore with potentially lower submission overhead. See Section 2.3.1.1 for more detail.

HBM

High Bandwidth Memory: Main memory technology, consisting of stacked DRAM-dies. Section 2.1 offers more detail.

Intel DML

Intel Data Mover Library: A library presenting a high-level interface with the DSA. View the usage example in Section 2.4 or the library documentation [10] for further information.

Memory Pressure

Memory Pressure: Situation where high memory utilization is encountered.

Mutating Method

Mutating Method: Member function (method) capable of mutating data of a classes instances. This is the default in C++, unless explicitly disabled by marking a method const or static.

Node

NUMA-Node: A Node in a NUMA-system. See the Entry for NUMA for an explanation of both.

NUMA

Non-Uniform Memory Architecture: Describes a system architecture organized into different Nodes with each node observing different access patterns to memory for the available address range.

NVRAM

Non-Volatile RAM: Main memory technology which, unlike DDR-SDRAM, retains data when without power.

QdP

Query-driven Prefetching: Methodology to determine database columns worth prefetching to cache in order to accelerate a queries. Described in Section 2.2.

SWQ

Shared Work Queue: A type of Work Queue to which submissions are implicitly synchronized, allowing safe usage from multiple processes. See Section 2.3.1.1 for more detail.

WQ

Work Queue: Architectural component of the DSA to which data is submitted by the user. See Section 2.3.1.1 for more detail on its function.

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] AMD, *High-Bandwidth Memory (HBM)*, `https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf`. (visited on 14th Feb. 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] 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`.

[7] 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).

[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] P. J. Denning, 'Virtual Memory', *ACM Computing Surveys*, vol. 28, no. 1, pp. 213–216, Mar. 1996. DOI: `10.1145/234313.234403`.

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

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

[12] 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).

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

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

[15] 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).

[16] Intel, *Intel® Xeon® CPU Max 9468 Processor*, `https://ark.intel.com/content/www/us/en/ark/products/232596/intel-xeon-cpu-max-9468-processor-105m-cache-2-10-ghz.html`. (visited on 14th Feb. 2024).

[17] 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).

[18] 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`.

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

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

[21] 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.`.

[22] cppreference.com, *CPP Reference Entry on std::atomic<T>::wait*, `https://en.cppreference.com/w/cpp/atomic/atomic/wait`. (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).