Bachelors Thesis

Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Anatol Constantin Fürst

21st January 2024

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

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

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Aufgabenstellung für die Anfertigung einer Bachelor-Arbeit

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

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

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

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

Prof. Dr.-Ing. Horst Schirmeier
Betreuender Hochschullehrer

Statement of Authorship

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

Dresden, 21st January 2024

Anatol Constantin Fürst

Abstract

...abstract ...

write the abstract

Contents

List of Figures

List of Tables

1 Introduction

write this chapter

2 Technical Background

write introductory paragraph

2.1 High Bandwidth Memory

High Bandwidth Memory is a novel memory technology promising an increase in peak bandwidth. It is composed of stacked DRAM dies [1, p. 1] and is slowly being integrated into server processors, notably the Intel® Xeon® Max Series [2]. High Bandwidth Memory (HBM) on these systems may be configured in different memory modes, most notably, HBM Flat Mode and HBM Cache Mode [2]. The former gives applications direct control, requiring code changes while the latter utilizes the HBM as cache for the systems DDR based main memory [2].

2.2 Query Driven Prefetching

write this section

2.3 Intel Data Streaming Accelerator

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

Introduced with the 4^{th} generation of Intel Xeon Scalable Processors, the DSA promises to alleviate the CPU from 'common storage functions and operations such as data integrity checks and deduplication' [4, p. 4]. To utilize the hardware optimally, knowledge of its workings is required. Therefore, we present an overview of the architecture, software, and the interaction of these two components, going into detail on the workings of the DSA engine itself. All statements are based on Chapter 3 of the Architecture Specification by Intel.

2.3.1 Hardware Architecture

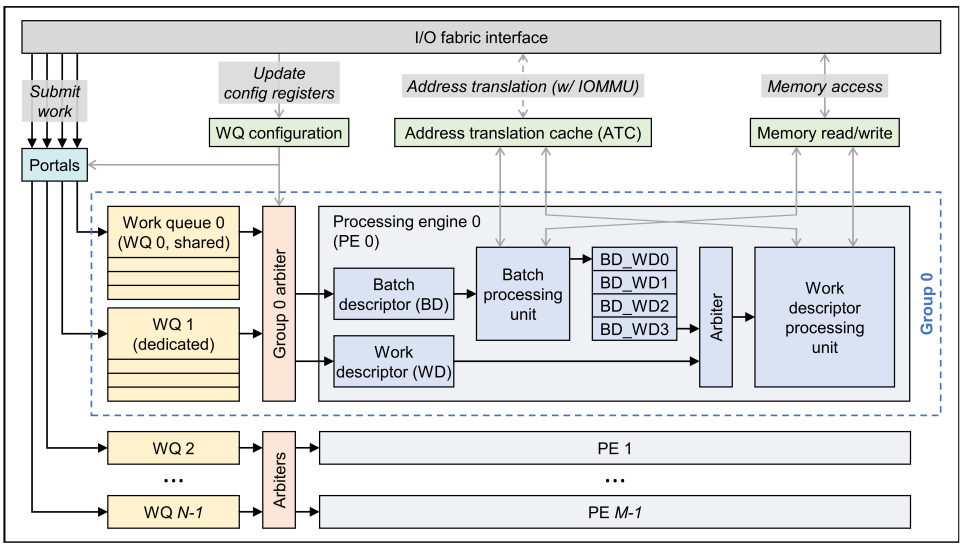

Figure 2.1: DSA Internal Architecture [5, Fig. 1 (a)]

The DSA chip is directly integrated into the processor and attaches via the I/O fabric interface over which all communication is conducted. Through this interface, it is accessible as a PCIe device. Therefore, configuration utilizes memory-mapped registers set in the devices Base Address Register (BAR). Through these, the devices' layout is defined and memory pages for work submission set. In a system with multiple processing nodes, there may also be one DSA per node, resulting in 4 being present on the previously mentioned Xeon Max CPU.

To satisfy different use cases, the layout of the DSA may be software-defined. The structure is made up of three components, namely Work Queue (WQ), Engine and Group. WQs provide the means to submit tasks to the device and will be described in more detail shortly. They are marked yellow in Figure 2.1. An Engine is the processing-block that connects to memory and performs the described task. The grey block of Figure 2.1 shows the subcomponents that make up an engine and the different internal paths for a batch or task descriptor. Using Groups, Engines and WQs are tied together, indicated by the dotted blue line around the components of Group 0 in Figure 2.1. This means, that tasks from one WQ may be processed from multiple Engines and vice-versa, depending on the configuration. This flexibility is achieved through the Group Arbiter, represented by the orange block in Figure 2.1, which connects the two components according to the user-defined configuration.

A WQ is accessible through so-called portals, light blue in Figure 2.1, which are mapped memory regions. Submission of work is done by writing a descriptor to one of these. A descriptor is 64 bytes in size and may contain one specific task (task descriptor) or the location of a task array in memory (batch descriptor). Through these portals, the submitted descriptor reaches a queue. There are two possible queue types with different

submission methods and use cases. The Shared Work Queue (SWQ) is intended to provide synchronized access to multiple processes and each group may only have one attached. A PCIe Deferrable Memory Write Request (DMR), which guarantees implicit synchronization, is generated via x86 Instruction ENQCMD and communicates with the device before writing [3, Sec. 3.3.1]. This may result in higher submission cost, compared to the Dedicated Work Queue (DWQ) to which a descriptor is submitted via x86 Instruction MOVDIR64B [3, Sec. 3.3.2].

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

Ordering of operations is only guaranteed for a configuration with one WQ and one Engine in a Group when submitting exclusively batch or task descriptors but no mixture. Even then, only write-ordering is guaranteed, meaning that 'reads by a subsequent descriptor can pass writes from a previous descriptor'. A different issue arises, when an operation fails, as the DSA will continue to process the following descriptors from the queue. Care must therefore be taken with read-after-write scenarios, either by waiting for a successful completion before submitting the dependant, inserting a drain descriptor for tasks or setting the fence flag for a batch. The latter two methods tell the processing engine that all writes must be committed and, in case of the fence in a batch, abort on previous error. [3, Sec. 3.9]

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

The completion of a descriptor may be signalled through a completion record and interrupt, if configured so. For this, the DSA 'provides two types of interrupt message storage: (1) an MSI-X table, enumerated through the MSI-X capability; and (2) a device-specific Interrupt Message Storage (IMS) table' [3, Sec. 3.7].

2.3.2 Software View

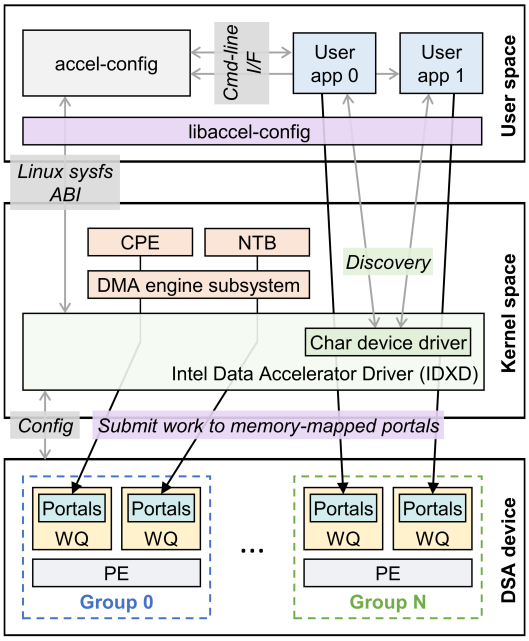

Figure 2.2: DSA Software View [5, Fig. 1 (b)]

Since Linux Kernel 5.10, there exists a driver for the DSA which has no counterpart in the Windows OS-Family [6, Sec. Installation], meaning that accessing the DSA is not possible to user space applications. To interface with the driver and perform configuration operations, Intel's accel-config [7] user space toolset may be used which provides a command-line interface and can read configuration files to set up the device as described previously, this can be seen in the upper block titled 'User space' in Figure 2.2. It interacts with the kernel driver, light green and labled 'IDXD' in Figure 2.2, to achieve this. After successful configuration, each WQ is exposed as a character device by `mmap` of the associated portal [5, Sec. 3.3].

Given the file permissions, it would now be possible for a process to submit work to the DSA via either MOVDIR64B or ENQCMD instructions, providing the descriptors by manually configuring them. This, however, is quite cumbersome, which is why Intel's Data Mover Library exists.

With some limitations, like lacking support for DWQ submission, this library presents a high-level interface that takes care of creation and submission of descriptors, some error handling and reporting. Thanks to the high-level-view the code may choose a different execution path at runtime which allows the memory operations to either be executed in hardware or software. The former on an accelerator or the latter using equivalent instructions provided by the library. This makes code based upon it automatically compatible with systems that do not provide hardware support. [6, Sec. Introduction]

2.3.3 Programming Interface

write this section

- choice is intel data mover library
- two concepts, state-based for c-api and operation-based c++
- just explain the basics (no code) and refer to dml documentation

2.4 System Setup and Configuration

write this section

Give the reader the tools to replicate the setup. Also explain why the BIOS-configs are required.

Setup Requirements:

- VT-d enabled
- limit CPUPA to 46 Bits disabled
- IOMMU enabled
- kernel with iommu and idxd driver support
- kernel option ”intel_iommu=on,sm_on”
- numa nodes for hbm access in bios

3 Performance Microbenchmarks

mention article by reese cooper here

write introductory paragraph

3.1 Benchmarking Methodology

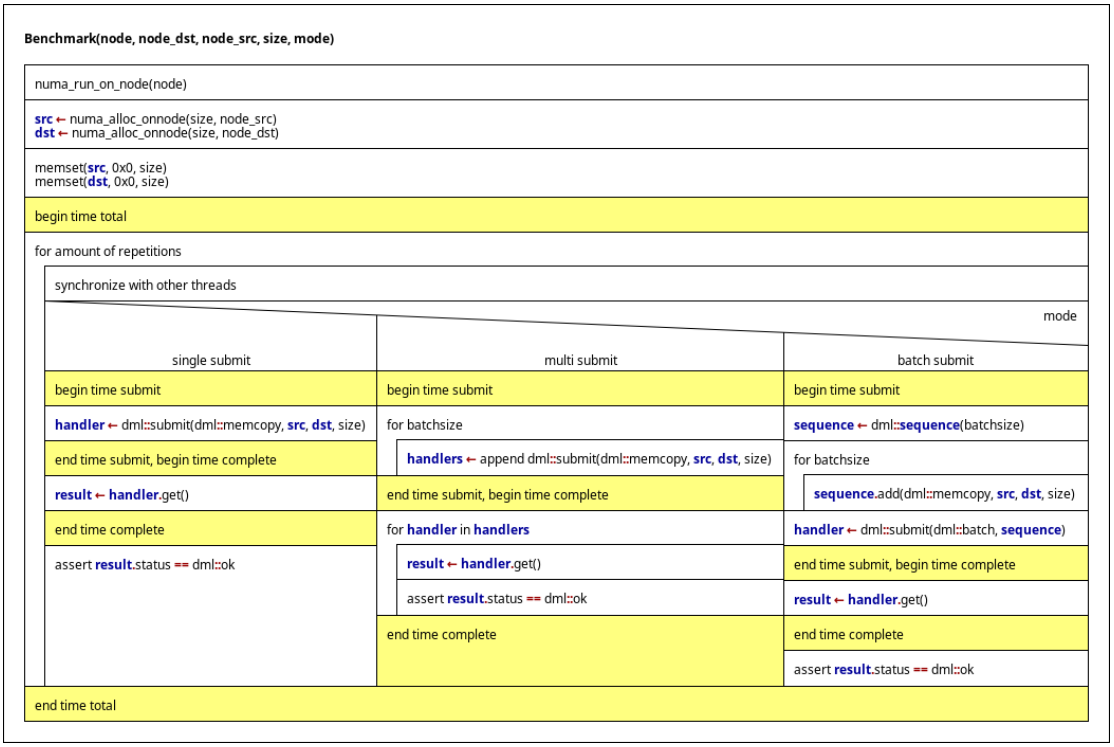

Figure 3.1: Throughput for different Submission Methods and Sizes

split graphic into multiple parts for the three submission types

3.2 Submission Method

With each submission, descriptors must be prepared and sent off to the underlying hardware. This is expected to come with a cost, affecting throughput sizes and submission methods differently. By submitting different sizes and comparing batching, single submission and utilizing the DSAs queue with multi submission we will evaluate at which data size which submission method makes sense.

write this section

In Figure 3.2 we conclude that with transfers of 1 MiB and upwards, the submission method makes no noticeable difference. For smaller transfers the performance varies

maybe remeassure with 8 KiB, 16 KiB as sizes

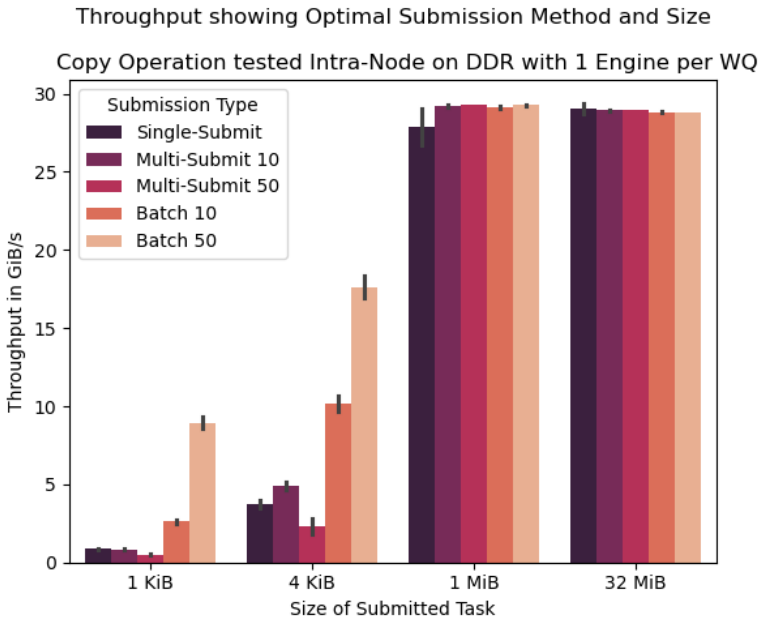

Figure 3.2: Throughput for different Submission Methods and Sizes

greatly, with batch operations leading in throughput. We assume that high submission cost of the SWQ cause all but the batch, which only performs one submission for its many descriptors, to suffer. This is aligned with the finding that 'SWQ observes lower throughput between 1-8 KB [transfer size]' [5, p. 6 and 7].

Another limitation may be observed in this result, namely the inherent throughput limit per DSA chip of close to 30 GiB/s. This is apparently caused by I/O fabric limitations [5, p. 5].

3.3 Multithreaded Submission

write this section

- effect of mt-submit, low because SWQ implicitly synchronized, bandwidth is shared
- show results for all available core counts
- only display the 1engine tests
- show combined total throughput
- conclude that due to the implicit synchronization the sync-cost also affects 1t and therefore it makes no difference, bandwidth is shared, no guarantees on fairness

write this section

3.4 Data Movement from DDR to HBM

write this section

- present two copy methods: smart and brute force
- show graph for ddr->hbm intranode, ddr->hbm intrasocket, ddr->hbm intersocket
- conclude which option makes more sense (smart)
- because 4x or 2x utilization for only 1.5x or 1.25x speedup respectively
- maybe benchmark smart-copy intersocket in parallel with two smart-copies intra-socket VS. the same task with brute force

3.5 Analysis

write this section

- summarize the conclusions and define the point at which dsa makes sense
- minimum transfer size for batch/nonbatch operation
- effect of mtsubmit -> no fairness guarantees
- usage of multiple engines -> no effect
- smart copy method as the middle-ground between peak throughput and utilization
- lower utilization of dsa is good when it will be shared between threads/processes

4 Design

write introductory paragraph

4.1 Detailed Task Description

write this section

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

4.2 Cache Design

The task of prefetching is somewhat aligned with that of a cache. As a cache is more generic and allows use beyond Query Driven Prefetching, the choice was made to solve the prefetching offload by implementing an offloading `Cache`. When referring to the provided implementation, `Cache` will be used from now on. The interface with `Cache` must provide three basic functions: requesting a memory block to be cached, accessing a cached memory block and synchronizing cache with the source memory. The latter operation comes in to play when the data that is cached may also be modified, requiring the entry to be updated with the source or the other way around. Due to the many possible setups and use cases, 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 while being removed when memory pressure due to restrictive memory size drives the `Cache` to flush unused entries.

4.2.1 Interface

To allow rapid integration and ease developer workload, a simple interface was chosen. As this work primarily focuses on caching static data, the choice was made only to provide cache invalidation and not synchronization. Given a memory address, `Cache::Invalidate` will remove all entries for it. The other two operations are provided in one single function, which we shall call `Cache::Access` henceforth, receiving a data pointer and size it takes care of either submitting a caching operation if the pointer received is not yet cached or returning the cache entry if it is. The cache placement and assignment of the task to accelerators are controlled by the user. In addition to the two basic operations outlined before, the user also is given the option to flush the cache using `Cache::Flush` of unused elements manually or to clear it completely with `Cache::Clear`.

As caching is performed asynchronously, the user may wish to wait on the operation. This would be beneficial if there are other threads making progress in parallel while the current thread waits on its data becoming available in the faster cache, speeding up local computation. To achieve this, the `Cache::Access` will return an instance of an object which from hereinafter will be referred to as `CacheData`. Through `CacheData::GetDataLocation` a pointer to the cached data will be retrieved, while also providing `CacheData::WaitOnCompletion` which must only return when the caching operation has completed and during which the current thread is put to sleep, allowing other threads to progress.

4.2.2 Cache Entry Reuse

When multiple consumers wish to access the same memory block through the `Cache`, we could either provide each with their own entry, or share one entry for all consumers. The first option may cause high load on the accelerator due to multiple copy operations being submitted and also increases the memory footprint of the system. The latter option requires synchronization and more complex design. As the cache size is restrictive, the latter was chosen. The already existing `CacheData` will be extended in scope to handle this by allowing copies of it to be created which must synchronize with each other for `CacheData::WaitOnCompletion` and `CacheData::GetDataLocation`.

4.2.3 Cache Entry Lifetime

By allowing multiple references to the same entry, memory management becomes a concern. Freeing the allocated block must only take place when all copies of a `CacheData` instance are destroyed, therefore tying cache entry lifetime to the lifetime of the longest living copy of the original instance. This makes access to the entry legal during the lifetime of any `CacheData` instance, while also guaranteeing that `Cache::Clear` will not have any unforeseen side effects, as deallocation only takes place when the last consumer has `CacheData` go out of scope or manually deletes it.

4.2.4 Usage Restrictions

As cache invalidation applies mainly to non-static data which this work does not focus on, two restrictions are placed on the invalidation operation. This permits drastically simpler cache design, as a fully coherent cache would require developing a thread safe coherence scheme which is outside our scope.

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

Secondly, invalidation is to be performed manually, requiring the programmer to remember which points of data are at any given point in time cached and invalidating them upon modification. No ordering guarantees will be given for this situation, possibly

leading to threads still having a pointer to now-outdated entries and continuing their progress with this.

Due to its reliance on libnuma for memory allocation and thread pinning, `Cache` will only work on systems where this library is present, excluding, most notably, Windows from the compatibility list.

4.2.5 Accelerator Usage

Compared with the challenges of ensuring correct entry lifetime and thread safety, the application of DSA for the task of duplicating data is simple, thanks partly to Intel Data Mover Library (Intel DML) [6]. Upon a call to `Cache::Access` and determining that the given memory pointer is not present in cache, work will be submitted to the Accelerator. Before, however, the desired location must be determined which the user-defined cache placement policy function handles. With the desired placement obtained, the copy policy then determines, which nodes should take part in the copy operation which is equivalent to selecting the Accelerators following 2.3.1. This causes the work to be split upon the available accelerators to which the work descriptors are submitted at this time. The handlers that Intel DML [6] provides will then be moved to the `CacheData` instance to permit the callee to wait upon caching completion. As the choice of cache placement and copy policy is user-defined, one possibility will be discussed in 5.

5 Implementation

write introductory paragraph

5.1 Locking and Usage of Atomics

The usage of locking and atomics proved to be the challenging. Their use is performance critical and mistakes may lead to deadlock. Therefore, they also constitute the most interesting part of the implementation which is why this chapter will focus extensively on the details of the implementation in regard to these.

5.1.1 Cache State Lock

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

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

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

5.1.2 CacheData Atomicity

The choice made in 4.2.2 requires thread safe shared access to the same resource. `std::shared_ptr<T>` provides a reference counted pointer, which is thread safe for the required operations, making it a prime candidate for this task. An implementation using it was explored but proved to offer its own set of challenges. As we wish to reduce time spent in a locked region, the task is only added to the nodes cache state when locked. Submission takes place outside, which is sensible, as this submitting one task should not hinder accessing another. To achieve the safety for `CacheData::WaitOnCompletion` outlined in 4.2.2 this would require the threads to coordinate which thread performs the actual waiting, as we assume the handlers of Intel DML to be non-threadsafe. In order to avoid queuing multiple of the same copies, the task must be added before submission. This results in a `CacheData` instance with invalid cache pointer and no handlers to wait for being available, requiring additional usage of synchronization primitives. With using `std::shared_ptr<T>` also comes the uncertainty of relying on the implementation to be performant. The standard does not specify whether a lock-free algorithm is to be used and [8] suggests abysmal performance for some implementations, although the full article is in Korean. No further research was found on this topic.

It was therefore decided to implement atomic reference counting for `CacheData` which means providing a custom constructor and destructor wherein a shared (through a standard pointer however) atomic integer is either incremented or decremented using atomic fetch sub and add operations [9] to increase or deacrease the reference counter and, in case of decrease in the destructor signals that the destructor is called for the last reference, perform actual destruction. The invalid state of `CacheData` achievable is also avoided. To achieve this, the waiting algorithm requires the handlers to be contained in an atomic pointer and the pointer to the cache memory be atomic too. Through this we may use the atomic wait operation which is guaranteed by the standard to be more efficient than simply spinning on Compare-And-Swap [10]. Some standard implementations achieve this by yielding after a short spin cycle [11].

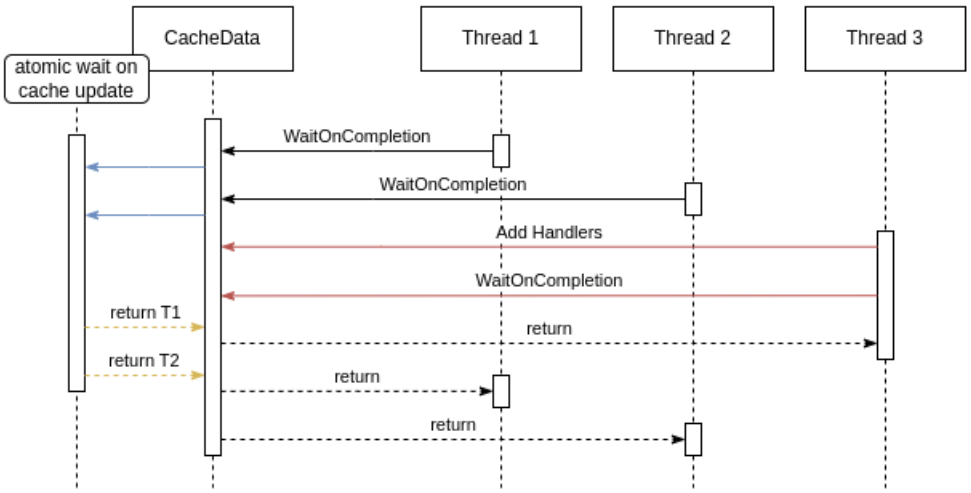

Figure 5.1: Sequence diagram for threading scenario in `CacheData::WaitOnCompletion`

Designing the wait to work from any thread was complicated. In the first implementation, a thread would check if the handlers are available and if not atomically wait [10] on a value change from nullptr. As the handlers are only available after submission, a situation could arise where only one copy of `CacheData` is capable of actually waiting on them. Lets assume that three threads T_1, T_2 and T_3 wish to access the same resource. T_1 now is the first to call `CacheData::Access` and therefore adds it to the cache state and will perform the work submission. Before T_1 may submit the work, it is interrupted and T_2 and T_3 obtain access to the incomplete `CacheData` on which they wait, causing them to see a nullptr for the handlers but invalid cache pointer, leading to atomic wait on the cache pointer (marked blue lines in 5.1). Now T_1 submits the work and sets the handlers (marked red lines in 5.1), while T_2 and T_3 continue to wait. Now only T_1 can trigger the waiting and is therefore capable of keeping T_2 and T_3 from progressing. This is undesirable as it can lead to deadlocking if by some reason T_1 does not wait and at the very least may lead to unnecessary delay for T_2 and T_3 if T_1 does not wait immediately.

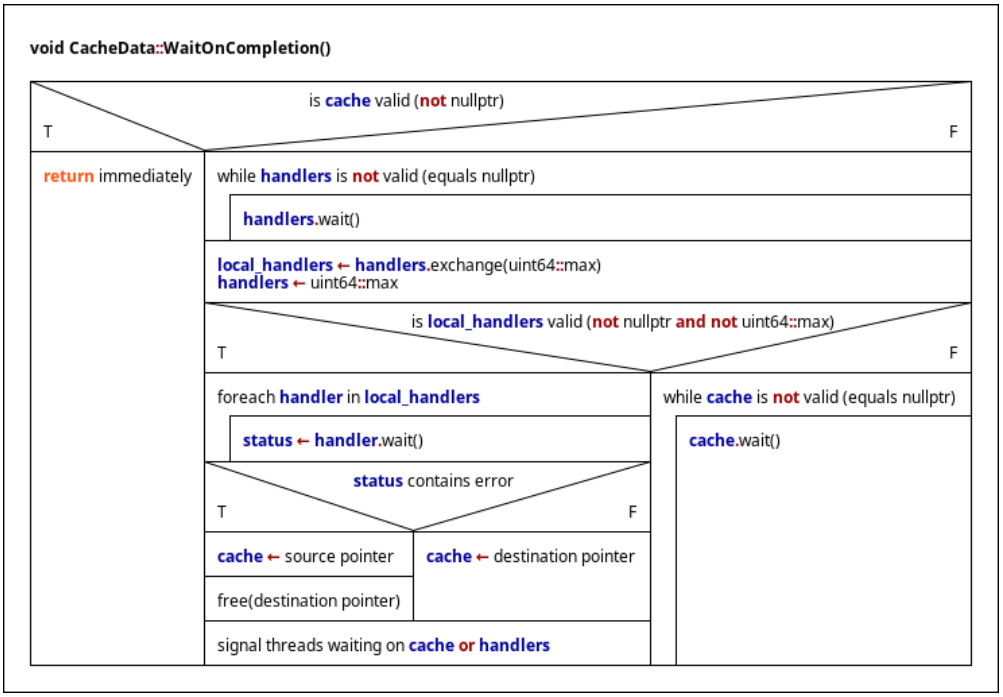

Figure 5.2: Code Flow Diagram for `CacheData::WaitOnCompletion`

To solve this, a different and more complicated order of waiting operations is required. When waiting, the threads now immediately check whether the cache pointer contains a valid value and return if it does, as nothing has to be waited for in this case. Let's take the same example as before to illustrate the second part of the waiting procedure. T_2 and T_3 now both arrive in this latter section as the cache was invalid at the point in time when waiting was called for. They now atomically wait on the handlers pointer to change, instead of doing it the other way around as before. Now when T_1 supplies the handlers, it also uses `std::atomic<T>::notify_one` [12] to wake at least one thread waiting on value change of the handlers pointer, if there are any. Through this the exclusion that was observable in the first implementation is already avoided. If nobody is waiting, then the handlers will be set to a valid pointer and a thread may pass the atomic wait instruction later on. Following this wait, the handlers pointer is atomically exchanged [13] with nullptr, invalidating it. Now each thread again checks whether it has received a valid local pointer to the handlers from the exchange, if it has then the atomic operation guarantees that is now in sole possession of the pointer. The owning thread is tasked with actually waiting. All other threads will now regress and call `CacheData::WaitOnCompletion` again. The solo thread may proceed to wait on the handlers and should update the cache pointer.

Some two additional cases must be considered for the latter implementation to be safe. The wait operation first checks for a valid cache pointer and then waits on the handlers becoming valid. After processing the handlers, they are deleted and the pointer therefore invalidated. Should the cache pointer now be invalid as well, deadlocks would ensue. Therefore, the thread which exchanged the handlers pointer for a valid local copy

must set the cache pointer to a valid value. Should one of the offloaded operations have failed, using the cache pointer is out of question as the datum it references might be invalid. The cache is set to the source address in this case. Secondly, after one thread has exchanged the pointer locally, threads may collect waiting on the handlers to become available. This can happen when the wait on the handlers takes sufficient amount of time during which both handlers and cache pointer are invalid. After waiting, the responsible thread must therefore signal all [14] threads waiting on the handler to continue.

Two types of deadlocks were encountered during testing and have been accounted for. On one hand, it was found that the guarantee of `std::atomic<T>::wait` to only wake up when the value has changed [10] is stronger than the promise of waking up all waiting threads with `std::atomic<T>::notify_all` [14]. The value of the handler pointer may therefore not be exchanged with nullptr which is the value we wait on. As the highest envisioned address requires the lower 52-bits of current 64-bit wide systems [15, p. 120] [16, p. 4-2] setting all bits of a 64-bit-value yields an invalid pointer which is used as the second invalid state possible. The second type was encountered when after creating a `CacheData` instance it was determined this exists in the cache already and dropped. As destruction waits on completion in order to ensure that no further jobs require the memory held, a deadlock would arise from the cache and handler pointers both being null and no handlers ever being set due to the instance being deleted immediately. To circumvent this, the constructor of `CacheData` was modified to point to source memory by default. Only after calling a separate initialization function will `CacheData` replace this with nullptr, therefore readying the instance for multithreaded usage.

5.1.3 Performance Guideline

Atomic operations come with an added performance penalty. No recent studies were found on this, although we assume that the findings of Hermann Schweizer et al. in “Evaluating the Cost of Atomic Operations on Modern Architectures“ [17] still hold true for today’s processor architectures. Due to the inherent cache synchronization mechanisms in place [17, Subsection IV.A.3 Off-Die Access], they observed significant access latency increase depending on whether the atomic variable was located on the local core, on a different core on the same chip or on another socket [17, Fig. 4]. Reducing the cost of atomic accesses would require a less generic implementation, reducing some of the guarantees we give in 4.2. This would allow reducing the amount of atomics required but is outside the scope of this work.

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

5.2 Accelerator Usage

After 4.2.5 the implementation of `Cache` provided leaves it up to the user to choose a caching and copy method policy which is accomplished through submitting function pointers at initialization of the `Cache`. In 2.4 we configured our system to have separate NUMA-Node (Node)s for accessing HBM which are assigned a Node-ID by adding eight to the Nodes ID of the Node that physically contains the HBM. Therefore, given Node 3 accesses some datum, the most efficient placement for the copy would be on Node $3 + 8 == 11$. As the `Cache` is intended for multithreaded usage, conserving accelerator resources is important, so that concurrent cache requests complete quickly. To get high per-copy performance while maintaining low usage, the smart-copy method is selected as described in 3.4 for larger copies, while small copies will be handled exclusively by the current node. This distinction is made due to the overhead of assigning the current thread to the selected nodes, which is required as Intel DML assigns submissions only to the DSA engine present on the node of the calling thread [6, Section "NUMA support"]. No testing has taken place to evaluate this overhead and determine the most effective threshold.

6 Evaluation

...evaluation ...

write this chapter

7 Conclusion And Outlook

write introductory paragraph

7.1 Conclusions

write this section

7.2 Future Work

write this section

- analyse whether prefetching yields better performance than hbm caching mode [2]
- evaluate impact of lock contention and atomics on performance
- provide optimized use case specific versions with less locking
- extend the cache implementation use cases where data is not static

Glossary

A

ATC

... desc ...

B

BAR

... desc ...

D

DMR

... desc ...

DSA

... desc ...

DWQ

... desc ...

E

Engine

... desc ...

ENQCMD

... desc ...

G

Group

... desc ...

H

HBM

... desc ...

I

Intel DML

... desc ...

IOMMU

... desc ...

M

MOVDIR64B

... desc ...

N

Node

... desc ...

P

PASID

... desc ...

S

SWQ

... desc ...

W

WQ

... desc ...

Bibliography

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

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

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

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

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

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

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

[8] 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..

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

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

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

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

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

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

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

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

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