# Enabling FPGA Memory Management for Big Data Applications Using Fletcher

L. Wijtemans

Master's thesis

Delft University of Technology

Challenge the future

# Enabling FPGA Memory Management for Big Data Applications Using Fletcher

by

**L. Wijtemans**

in partial fulfillment of the requirements for the degree of

**Master of Science**
in Embedded Systems

at the Delft University of Technology,
to be defended publicly on Tuesday August 27, 2019 at 10:30 AM.

| | | |
|---|---|---|
| Student number: | 1505378 | |
| Faculty: | Electrical Engineering, Mathematics and Computer Science | |
| Department: | Quantum & Computer Engineering | |
| Programme: | Embedded Systems | |
| Specialisation: | Computer Architecture | |
| Project duration: | November 26, 2018 – August 23, 2019 (40 ECTS) | |
| Thesis committee: | Dr.ir. Z. Al-Ars, | TU Delft, *supervisor* |
| | Prof.dr. H.P. Hofstee, | TU Delft, IBM Austin |
| | Dr. J.S. Rellermeyer, | TU Delft |
| | Ir. J.W. Peltenburg, | TU Delft |

# Abstract

Availability of FPGAs is increasing due to cloud service offerings. In the wake of a new in-memory storage format specification, Apache Arrow, FPGAs are increasingly interesting for data processing acceleration in the big data domain. The Fletcher framework can be used to easily develop FPGA accelerated applications that access data stored in Apache Arrow format, while providing throughput near the system's limit. The current implementation of Fletcher has limited support for memory management of FPGA-local memory, with one of the biggest limitations being that memory can only be used once.

This thesis explores several memory management techniques which could be suitable for use on FPGAs in a big data context. Paged memory is implemented on FPGA within the Fletcher framework in order to facilitate this memory management. The implemented system takes less than 5 % of a data centre FPGA card's resources (Xilinx UltraScale+ VU9P). Experiments show that the paged memory provides throughput of over 99.7 % of the system's throughput for linear memory accesses. Random memory access throughput for paged memory drops to between 30 % and 90 % of the system's original throughput, depending on request size. The performance drop can be lightened or even prevented by employing suitable address-translation caches.

# Preface

You are about to read the concluding work of a nine month project. I am thankful for all the support that I have gotten over this period from my mother and father, without whom I would not have been able to pursue my studies in this way. I am grateful for the ever unwavering support from my grandfather, who could not help but be constantly curious about those upcoming quantum computers.

Special thanks go to Johan Peltenburg for offering the project in the first place. He was not only always willing and able to help with Fletcher internals, but also brought much welcome interesting conversations during lunch. Additionally, I wish to thank Jeroen van Straten and Matthijs Brobbel for their comments on my initial plans, and Jian Fang for his paper suggestions.

I want to thank Zaid Al-Ars for giving me the opportunity to work on an interesting project like this, and for agreeing to be my thesis supervisor. Additional thanks go out to Zaid Al-Ars and Peter Hofstee for their helpful guidance and insights to give the project a good start. All members of the thesis committee get my thanks for their time, Zaid Al-Ars, Peter Hofstee, Jan Rellermeyer, and Johan Peltenburg.

Finally, I want to thank my friends and family, who may not have seen me as much as they wanted or needed during this period. I'm glad I was permitted the time it took to undertake this project, and surely I'll repay most of you.

In loving memory of Dirk Rietveld.

*Lars Wijtemans*
*Spijkenisse, August 2019*

This thesis is typeset with XƎLATEX using TU Delft house style fonts. Diagrams are created with Dia. Graphs are plotted with PGFPlots.

# Contents

# Acronyms

**AWS** Amazon Web Services. 5, 11, 17, 21, 27, 33

**AXI** Advanced eXtensible Interface. 7, 13, 34

**BRAM** Block Random Access Memory. 7, 21, 34

**CAM** Content Addressable Memory. 18

**CAPI** Coherent Accelerator Processor Interface. 5, 12

**CLB** Configurable Logic Block. 28, 34

**CPU** Central Processing Unit. 1, 12

**DRAM** Dynamic Random-Access Memory. 7, 21, 27, 34

**FIFO** First In, First Out. 18

**FPGA** Field-Programmable Gate Array. 1, 2, 3, 5, 6, 7, 9, 11, 12, 13, 17, 18, 21, 27, 28, 34

**GPGPU** General-Purpose Graphics Processing Units. 1

**HBM** High Bandwidth Memory. 7

**IP** Intellectual Property. 7

**MMIO** Memory-Mapped Input Output. 18

**RAM** Random Access Memory. 11

**TLB** Translation Lookaside Buffer. 6, 18

**VHDL** VHSIC[1] Hardware Description Language. 7, 17

**VHSIC** Very High Speed Integrated Circuit. ix, 7

[1]Very High Speed Integrated Circuit

# 1 Introduction

## 1.1. Context

FPGAs[1] are a valuable tool in accelerating certain classes of data processing, able to compete with GPGPU[2] based acceleration on latency, throughput, and performance per Watt for certain applications. In the last two to three years, cloud providers started to include FPGAs as part of their offering. This takes away the entrance barrier of buying (expensive) FPGAs, bringing FPGA acceleration within reach of anyone with the expertise to program them. However, a big obstacle in using FPGAs is that they are more difficult to program than conventional CPUs[3], requiring specifically trained personnel, and have longer development cycles.

Some tools and frameworks, like high level synthesis and *SDAccel*, exist to ease the development burden of FPGAs. These tools can be very helpful when implementing algorithms on FPGAs, but do not always produce implementations that perform well. One of the cases where these automated tools fall short is in accessing data stored in an upcoming big data storage standard, *Apache Arrow*. The Apache Arrow data storage format is a specification for contiguous in-memory data storage. [1] The standard exists to increase interoperability between applications and reduce data (de)serialisation costs.

To provide a performant and resource efficient way of accessing data stored in the Apache Arrow format from an FPGA, the Fletcher framework is being developed at the TU Delft. [2] This framework provides an interface between the data, stored in the Apache Arrow format, and a computation kernel on an FPGA. The computation kernel is application specific and is separate from the framework for accessing the data. By using Fletcher, developers can focus on creating the computation kernel, instead of spending time designing data fetching and transforming logic.

Internally, Fletcher consists of hardware building blocks for prefetching data, bus multiplexers, and host-side software to interact with the FPGA. An overview of Fletcher can be found in Figure 1.1. The system uses input that is stored in the Apache Arrow format, feeds it through application specific processing logic, and produces output that is structured according to the Apache Arrow format. This input and output data is stored in a memory that can be located on the host system or on the FPGA board itself. The host has an operating system and libraries to manage the allocation of its memory, but the FPGA does not. In the current implementation of the Fletcher framework, data that is allocated on FPGA board memory is put there contiguously by the host software, without a mechanism to re-use a memory range once specific data is no longer needed. Another problem is that the size of generated output data is not generally known in advance, in which case allocated output memory may either not be large enough to accommodate all the generated data, or is over-dimensioned.

[1]Field-Programmable Gate Arrays
[2]General-Purpose Graphics Processing Units
[3]Central Processing Units

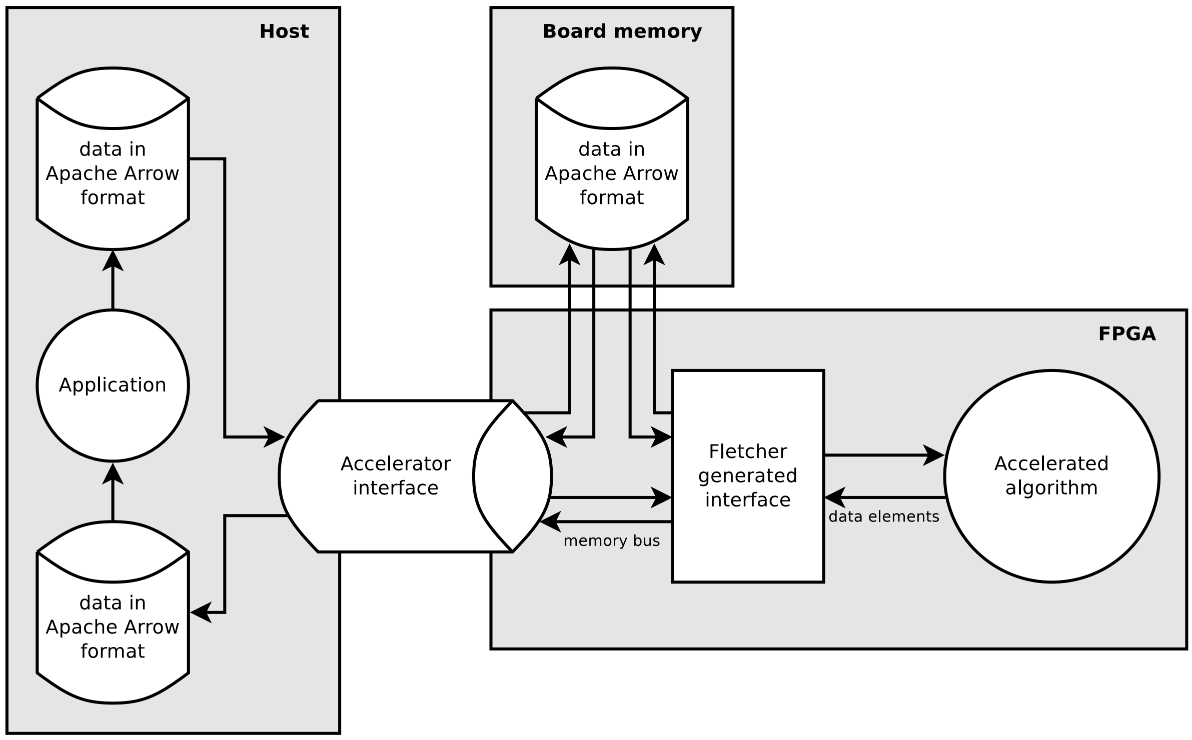

Host
data in Apache Arrow format
Application
data in Apache Arrow format
Board memory
data in Apache Arrow format
FPGA
Accelerator interface
Fletcher generated interface
Accelerated algorithm
memory bus
data elements

Figure 1.1: High level overview of Fletcher

## 1.2. Problem definition

Consider an example application, illustrated in Figure 1.2, that (i) uses an input dataset, A, of size $S_A$, and (ii) splits it into two datasets, B and C, of sizes $S_B$ and $S_C$, where $S_B + S_C = S_A$. The total needed space for all three datasets is $2S_A$. However, since the size of dataset B and C can be anywhere between 0 and $S_A$, a total of $3S_A$ needs to be allocated to run this application. Furthermore, when (iii) datasets B and C are processed further on the FPGA to produce dataset D, for which dataset A is not needed, the space previously occupied by dataset A cannot be used to store dataset D. Finally (iv), datasets should be able to be split in memory so that free space fragmentation does not prevent further allocations, when there is enough free memory in total.

From the problem description, the following research questions can be formulated:

1. Which memory management methods can provide the required functionality?
   (a) Allocate memory for buffers.
   (b) Reuse memory from buffers that are no longer in use.
   (c) Allow allocations to grow as required.
   (d) Allow usage of multiple separate memories.
   (e) Allow usage of host memory for platforms that support it.
   (f) Present a malloc-like interface.
   (g) Allow random access into the buffers.
2. Which memory management method is the most appropriate, taking into account
   (a) performance impact,
   (b) memory usage overhead,
   (c) implementation complexity?
3. Can a suitable memory management method be implemented within Fletcher?
4. What are optimal parameters for the implemented memory management method?

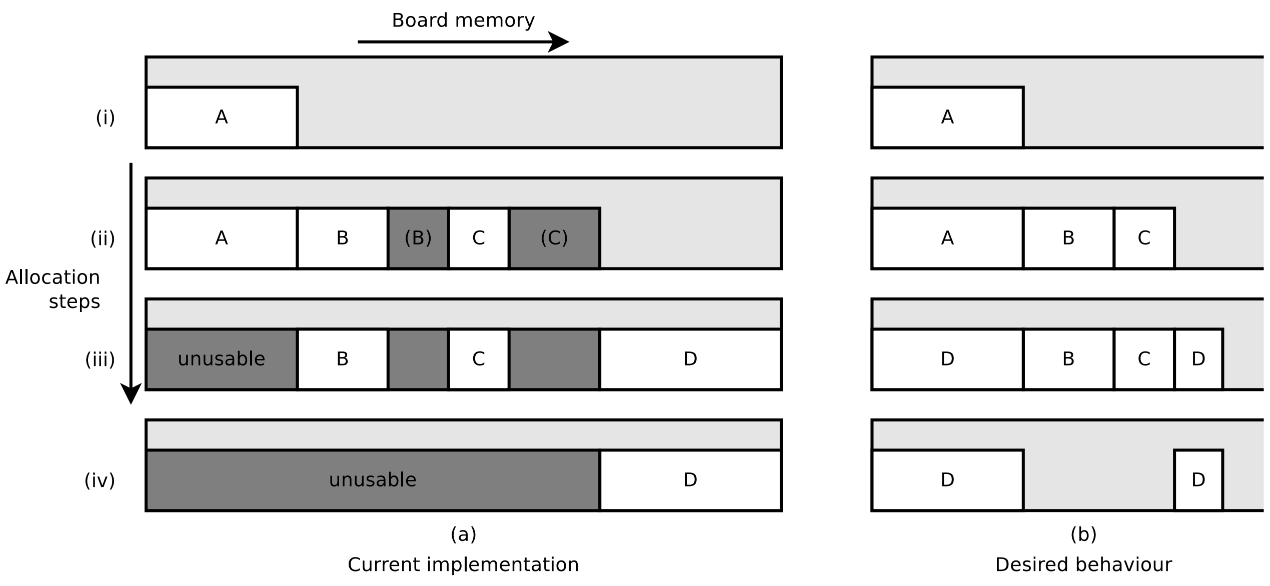


Figure 1.2: Current and desired memory allocation pattern

5. What is the cost of the implemented memory management method in terms of
   (a) performance,
   (b) memory usage overhead,
   (c) FPGA area?

## 1.3. Thesis outline

Chapter 2 will introduce Fletcher in some more depth, in addition to some concepts from memory management. Three alternative approaches to the memory management challenge will be discussed in Chapter 3. An architecture for implementing memory management in Fletcher is established in Chapter 4, after which the implementation is explained in Chapter 5. The performance impact of several implementation parameters are explored in Chapter 6. Finally, the main findings and answers to the research questions are discussed in Chapter 7.

# 2 Background

## 2.1. Fletcher

Fletcher accesses data that is stored in the Apache Arrow format and provides it to a user provided FPGA hardware design. The data format stores data in one or more *buffers*, which are contiguous sections of memory. The individual buffers are typically read linearly, from beginning to end. However, some applications may require random access into these buffers. When data is generated and stored, buffers need to be allocated before the data can be written to memory. However, it is not always known in advance how much data is stored in each buffer. It can therefore happen that a buffer needs to be made bigger after it has been allocated. Buffers can of course be over-dimensioned, but this would cause a big memory overhead.

Different platforms with differing FPGA accelerator interfaces exist. The two main platforms supported by Fletcher are OpenPOWER's CAPI[1], and AWS[2]' FPGA instances with a PCIe interface. CAPI takes care of virtual address translation for host memory, allowing direct access to host memory from the FPGA. The AWS system does have access to host memory from the FPGA, but does not perform virtual address translation. This makes direct usage of the host memory difficult, since address translation needs to be done by the user, in addition to taking extra precautions to prevent virtual to physical mappings from chaning, like memory pinning.

## 2.2. Memory management

Memory management has been a challenge in computer systems for a long time and a lot of effort has been spent to make memory allocation efficient, to varying degrees of success [3]. A lot of focus has been on heap memory management, since this has been the most performance critical in the past [4] [5] [6]. In the present, big data applications have very different memory needs than has been usual. They need a lot more memory, and use the memory in a different way: usually scanning through an entire dataset from beginning to end, instead of adhering to the classical working set model [7].

An important measure for the effectiveness of a memory management technique is internal fragmentation and external fragmentation. Internal fragmentation is the amount of wasted memory due to the fact that allocation sizes are rounded up towards a certain size. External fragmentation is fragmentation of the free space, and can cause contiguous allocations to fail even when enough free space is available in total.

## 2.3. Virtual memory on FPGAs

Virtual memory has already been employed on FPGAs before. CAPI allows an FPGA access to the processor's virtual address space. [8] The virtual to physical translation is done in much the same way as a processor does when accessing virtual memory. A big advantage of this system is, is that host memory can be directly accessed from the FPGA, without having to copy any data to the FPGA's

[1]Coherent Accelerator Processor Interface
[2]Amazon Web Services

memory. A feature that still misses from this is the ability to use the FPGA's on-board memory within the same virtual address space.

The Intel Core Cache Interface provides virtual to physical address translation by copying the host's page tables to the FPGA and using an FPGA based page table walker. [9] To increase performance, the FPGA uses a TLB[3] with a level 1 and level 2 cache of 512 entries. The system supports 4 KiB and 2 MiB pages. This also misses the ability to use the FPGA's on-board memory within the same virtual address space.

[3]Translation Lookaside Buffer

# 3 Alternative solutions

## 3.1. Requirements

The memory management system must have a *malloc*-like interface and behaviour. Meaning that for allocations, a certain size is requested, and the system returns a pointer to the start of the allocated buffer. For deallocations, only the pointer that was returned previously is given. So the memory management system is responsible for keeping track of the size of each allocation.

Fletcher will read from and/or write to one or more buffers, which are contiguous data sections in memory. Each of these buffers will commonly be accessed in a sequential manner, but in some cases may be accessed randomly. Fletcher is mainly used in big data environments, where datasets are generally large and throughput is the most important measure of performance.

The system should integrate with Fletcher using Fletcher's internal AXI[1]-like bus with minimal modifications to Fletcher itself. Additionally, the system should be able to be integrated in Fletcher generated designs programmatically in the future, although implementing this is outside the scope of this project.

When writing to a buffer, the total amount of data written may exceed the size of the allocated memory. In this case, it must be possible to grow the size of the allocated memory without losing previously written data.

FPGAs sometimes have multiple memories. For instance multiple DRAM[2] channels, internal BRAM[3], and/or a special HBM[4]. It should be possible for applications to choose a specific memory for each buffer.

Since one of Fletcher's goals is to be vendor-agnostic and IP[5]-free, these additions should also use no specific IP or vendor specific primitives. Since Fletcher is written in VHDL[6], and include tools that mainly process VHDL, the memory management system should also be written in VHDL.

## 3.2. Heap based management

One of the more direct approaches is to treat the FPGA's memories as heaps, much like heap memory is managed in C programs. A big advantage of this method is that once memory is allocated, physical addresses can be used directly for accessing the memory. This means that there will be no memory performance decrease when accessing the allocated buffers. However, heap based allocation algorithms often target applications that make a lot of smaller (order of 1 MiB) allocations and use trees and linked lists to keep track of allocated space [10]. Using it for large allocations may cause allocations to fail due to external fragmentation.

With this kind of management, growing existing allocations is usually necessarily done by copying the allocation to another physical memory location. When a buffer is filled linearly with $w$ bytes and

[1]Advanced eXtensible Interface
[2]Dynamic Random-Access Memory
[3]Block Random Access Memory
[4]High Bandwidth Memory
[5]Intellectual Property
[6]VHSIC Hardware Description Language

needs to grow each time it is full, the total amount of data written to physical memory for that buffer is given by

$$t = w + \frac{d^{\lceil \log_d(w) \rceil} - 1}{d - 1}, \tag{3.1}$$

where $d$ is the ratio of the grown buffer size to the original size. This leads to a write amplification, $a = \frac{T}{w}$, for large buffers given by

$$\frac{w + \frac{w}{d-1}}{w} = 1 + \frac{1}{d-1} \leq a < 2 + \frac{1}{d-1} = \frac{w + \frac{dw}{d-1}}{w} \tag{3.2}$$

in addition to *reading* $\frac{w}{d-1}$ to $\frac{w}{d-1} + w$ bytes for copying. Choosing the growing ratio $d$ to be large will minimise copy overhead, but will also lead to greater potential internal fragmentation. The average fraction of memory wasted due to internal fragmentation is given by $f = \frac{1}{2}(d-1)$. When choosing the convenient factor $d = 2$, average memory usage overhead for growable buffers due to internal fragmentation will be 50 % of the size of the written data. Average write throughput will be decreased to 40 % of the baseline throughput due to write amplification, not taking into account the extra reads that are needed.

## 3.3. Segmented memory

Segmented memory is characterised by referring to memory by *segment* and offset. The segment can be a buffer identifier, so that each buffer lives in its own segment. This still leaves open in which way the segment and offset should be mapped to physical memory. A promising technique is using the physical memory like an extent based file system uses hard disk space: each segment is comprised of one or more extents (variable sized contiguous blocks), which are individually mapped to physical memory, illustrated in Figure 3.1. The mapping of extents can be done like heap based management. This technique is similar to the direct segment method proposed by Basu et al. [11]. Overall this method is much more flexible than pure heap based management. Since each extent can be placed at a different location in physical memory, a buffer does not need to be contiguous in physical memory like is the case with heap based management. This takes away the need to copy an entire buffer to a new location, so there is no write amplification with this method.

External fragmentation poses little problems, since extents can be made to fit any contiguous free space in physical memory. Internal fragmentation can be managed by using exact-fitting extents where possible, and growing allocations by a small amount each time. The biggest disadvantage of this method is the complexity of keeping track of free space (of variable sized blocks), splitting new allocations in case of fragmentation, and random access. Random access requires scanning of all the extent mappings of a segment, or traversal of some kind of lookup structure. The extent mappings could be cached, but the amount of mappings for a given buffer depends on the fragmentation state of the memory and is therefore difficult to estimate.

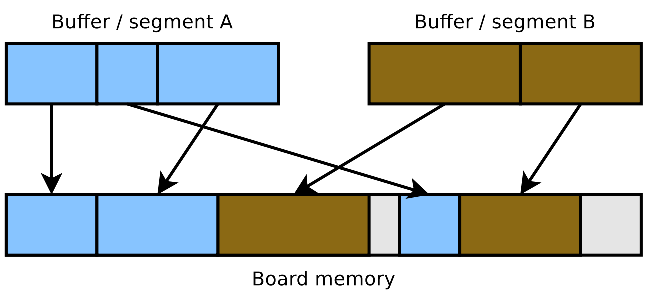


Figure 3.1: Example of segmented memory with extents

| method | write throughput | internal fragmentation overhead | random access | implementation |
|---|---|---|---|---|
| heap ($d = 2$) | <40 % | 50 % | direct | trees (complex) |
| segmented with extents | 100 % | up to 7 % (for growable) | scan / search | lists / trees (complex) |
| paged ($p = 4M$, 32 MiB $< b <$ 1 GiB) | 100 % | 0.2 % to 7 % | lookup | page table (simple) |

Table 3.1: Comparison of memory management techniques

## 3.4. Paged memory

Paged memory is often used in contemporary operating systems. For simplicity, this section constrains itself to paged memory with a single flat virtual address space. With paged memory, the virtual address space that is used by the allocations can be much larger than the available physical memory. The sheer size of the virtual memory can prevent external fragmentation from causing problems. There will always be a large enough contiguous free space in virtual memory for desired allocations.

The virtual memory is mapped to physical memory in fixed size blocks. The size of these blocks is called the page size, and is an important factor for internal fragmentation. Any allocation that is not a multiple of the page size will waste space, up to one page per allocation. This means the fraction of wasted memory ($f$) due to internal fragmentation depends on the relative size of allocations ($b$) to page size ($p$) and is given by $f = \frac{p}{2b}$. One can observe that a smaller page size leads to less wasted memory [12]. However, a smaller page size will also lead to more pages, which all need to be tracked.

Random access will cost a lookup in the page table, but does not require a scan or search as with variable sized extents.

## 3.5. Comparison

To evaluate the properties of the memory management techniques, certain assumptions are made. For the fragmentation estimation of paged memory, we will assume a 4 MiB page size and an average buffer size between 32 MiB and 1 GiB. The estimated write throughput, internal fragmentation overhead, and random access method are listed in Table 3.1. The *heap* method is unsuitable due to the low write throughput for dynamically growing buffers and problems with external fragmentation. The *segmented* method seems promising, but allocation algorithms get complex and the performance of random access is questionable. The internal fragmentation is comparable to the paged method for growable buffers and depends on the size the buffer grows by each time. Paging is relatively simple to implement through page tables, has acceptable internal fragmentation, has no problems with external fragmentation and has no inherent throughput limitations. Therefore, paging will be implemented for further evaluation.

### Host-based versus FPGA-based management

The allocation algorithms can be implemented either on the host or on the FPGA. The advantage of letting the host perform allocations, is that implementing an algorithm in software usually takes less development time than implementing the same algorithm in FPGA hardware. A disadvantage is that allocations initiated from the FPGA require at least a round trip communication delay, while these allocations are probably the most time critical compared to allocations initiated from the host, since they can occur while processing is happening and might stall writes from the FPGA. This disadvantage is exacerbated when using an allocation algorithm that requires frequent updates to allocations. For this reason, FPGA-based memory management is chosen.

# 4 Architecture

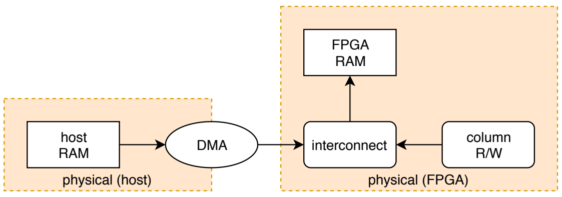
The required memory management operations are typically implemented by a software library that manages the application's heap memory. In fact, the requirements are constructed around the guarantees that the `malloc`, `realloc` and `free` library functions give. These libraries often have optimizations that target multi-threading, fragmentation, or a large amount of small allocations [13], but these are unnecessary in our particular application domain. What these libraries use under the hood for large allocations such as is typical in Fletcher, is the `mmap` system call. This system call is used to map RAM[1] into the application's virtual address space. Unfortunately, the FPGA does not have a virtual address space (yet). This will have to be implemented.

To implement a virtual address space, we will need a component that translates virtual addresses into physical addresses, the *translator*. Next to that, a component that manages the virtual and physical memory is needed, which here is called *allocator*. It will present a malloc-like interface, as specified in the requirements. Internally it will work like the `mmap` system call, updating page tables as required. [14]

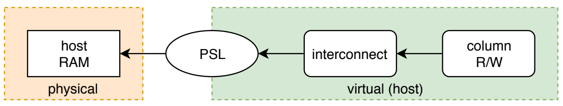
## 4.1. Memory address space

On systems like AWS FPGA instances, the virtual address space of the host and the address space of the FPGA are separate, as illustrated in Figure 4.1c. The virtual address space on the FPGA is accessible from the host through an abstraction provided by AWS software. It is then possible to use any arbitrary

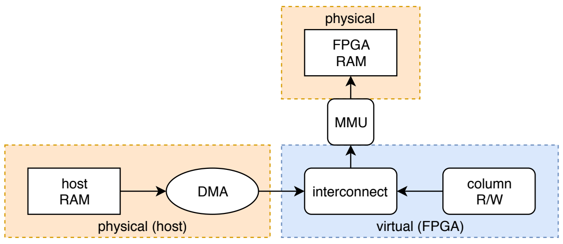
[1]Random Access Memory



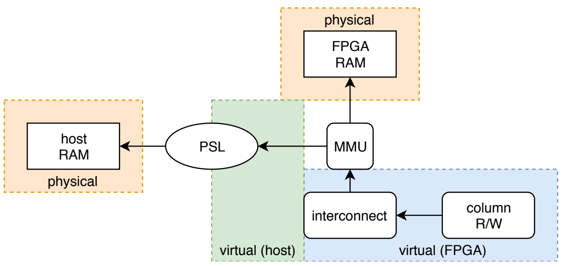
(a) Original AWS system



(b) Original CAPI system



(c) AWS system with virtual memory



(d) CAPI system with virtual memory

Figure 4.1: Original and virtualized conceptual memory organisation for CAPI systems and AWS systems

addressing space and scheme, since addresses that are used on the FPGA are unambiguously belonging to the FPGA memory. However, since systems using CAPI give the FPGA access to the host's virtual address space (Figure 4.1b), this is not possible on these systems. Note that the virtual address space that we will create on the FPGA and physical memory on it will not be directly accessible from the host system on this architecture (Figure 4.1d). In principle, the entire 64 bit address space refers to virtual memory on the host. Here a mechanism is needed to be able to tell apart virtual host addresses from virtual and physical FPGA addresses. Addresses of 65 bit were considered, but these do not fit in standard (64 bit) data structures or bus address lines. The chosen solution is to designate a portion of the host virtual address space for use on the FPGA. Current Power and x86_64 CPUs and operating system implementations do not use the full 64 bit address space, but only 48 to 57 bit, leaving a portion of the space available for use on the FPGA. When future processors and operating systems do use the full 64 bit address space, a fixed portion of it will have to be reserved through the operating system. This should be fine, since the available virtual address space on the host is very big [11].

## 4.2. High level overview

The main components of the design are (i) the address translator and page table walker, and (ii) the memory allocator. The address translation takes care of translating the bus requests which use virtual addresses to physical addresses, with the help of a page table walker. For special cases, the page table walker can defer address translation to the allocator. The memory allocator finds free virtual and physical memory to satisfy an allocation request, after which it will alter the page tables.

Attaching a dedicated translator to each buffer reader and writer allows it to take advantage of the fact that most buffers will be accessed linearly. When this is the case, only a single cache entry is needed in each translator, keeping the implementations compact. The two translators in Figure 4.2 could be placed after the bus arbiter, but the accesses as seen by the translator would no longer be strictly linear.

A page table walker can be shared between multiple translators. The page table walker can be pipelined, allowing it to perform multiple page table walks simultaneously. However, when a translation needs to be deferred to the allocator by the page table walker, the pipeline will need to stall until the allocator has serviced the request. For this reason, it may not be desirable to use only a single page table walker in the entire design.

The page table walker and the allocator respond to lookup requests, which consists only of a virtual address. The response includes the virtual address, the accompanying physical address, and a validity mask. The virtual address is included in the response to be able to broadcast gratuitous lookup results to, for instance, invalidate cache entries. The validity mask is used to convey the size of a contiguous virtual to physical mapping, allowing large mappings to be conveyed and cached efficiently, irrespective of the used page size.

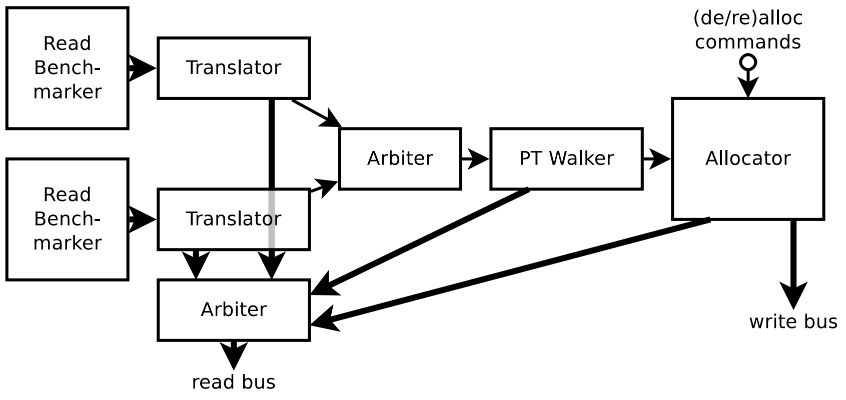


Figure 4.2: Structure of an example top level

## 4.3. Address translator

The address translator has the same AXI-like interface that is used by Fletcher and it is inserted on the address bus after a buffer reader or writer. The data bus is connected outside the translator without changes. If the input address is in the FPGA's virtual address space, the address is translated with the help of an optional local cache. If the input address is not found in the local cache, a lookup request will be sent to a page table walker, which may be shared between translators. The lookup result from the page table walker is then stored in the cache, together with the address validity mask. Finally, the physical address is passed on to the output of the address translator. If the input address falls outside of the FPGA's virtual address space, the input address is passed on unchanged, and no lookup requests are issued to the page table walker.

## 4.4. Page table walker

The page table walker accepts lookup requests consisting of a virtual address. It then walks the page table that is stored at a predefined location in memory, using Fletcher's memory bus. When the page table indicates there is no mapping to a physical address, the lookup is deferred to the allocator, otherwise the result of the page table walk is used. This facilitates the option to allocate physical memory for a virtual address only when it is first accessed. The page table walker responds with the original virtual address, the corresponding physical address, and the validity mask.

The page table walker may also perform a lookup for adjacent virtual addresses and adjust the validity mask accordingly when the address translation is valid for these adjacent addresses. For example, with a page size of 256 B, when a lookup for virtual address `0xABCD00` is requested, the page table walker may perform a lookup for addresses `0xABC000` through `0xABCF00`. Then the validity mask can be `0xFFF000` instead of `0xFFFF00` if for each of the resulting physical addresses ($A_{\text{phys}}$) and their corresponding virtual addresses ($A_{\text{virt}}$) it holds that $A_{\text{phys}} = (A_{\text{resp}} \text{ AND } M) \text{ OR } (A_{\text{virt}} \text{ AND NOT } M)$, where $M$ is the validity mask and $A_{\text{resp}}$ is the physical address provided by the page table walker in the response.

The page table walker should be able to absorb all outstanding bus requests. When it is unable to, a deadlock situation can occur where the bus is unusable because of the queued responses for the page table walker.

## 4.5. Allocator

The allocator is responsible for allocating virtual memory, physical memory, and updating the page tables accordingly. The interface for (de/re)allocation should be similar to the `malloc` family of memory management functions, meaning that an allocation request includes a desired size and results in a pointer to the start of the allocated memory. A *free* request includes only a pointer previously returned by the allocator in response to a (re)allocation request. A reallocation request includes both a previously returned pointer and a new desired size, to which a new pointer is returned in response. This means that the size of an allocation should be recorded internally, since it is not provided in a *free* request. Requests can be made by the host runtime software, the Fletcher framework, and user provided logic on the FPGA.

The allocator is not required to allocate physical memory immediately. A page table walker will forward a virtual to physical translation request to the allocator when the virtual address does not have a corresponding physical address. The allocator should then allocate physical memory, update the page tables, and respond to the page table walker with the physical address so that it does not need to walk the page tables again. The allocated physical address can be provided to the page table walker before the page table updates are visible to the page table walker, since reading the old state of the page tables will not lead to incorrect functioning.

The response of either allocation or *free* requests and that of reallocation requests should be given only after the corresponding page table updates are visible to all page table walkers. This is to ensure the page table walkers do not read a mapping that existed previously on the same virtual address.

Multiple readers and/or writers can be used to access a single buffer allocated by this system. However, care must be taken when these buffers are reallocated. From the time a reallocation (or *free*) request is issued to the allocator, the original base address for the buffer must be considered invalid and any accesses through this base address are illegal. This means that all readers and writers that use the base address that is going to be reallocated must be suspended and all outstanding read and write operations must have at least passed the address translator unit before a reallocation request can be issued safely. When the allocator responds to the reallocation request, the readers and writers can resume their requests with the new base address. This process is illustrated in Figure 4.3.

Just like the page table walker, the allocator should be able to absorb all outstanding bus requests. When it is unable to, a deadlock situation can occur where the bus is unusable because of the queued responses for the allocator.

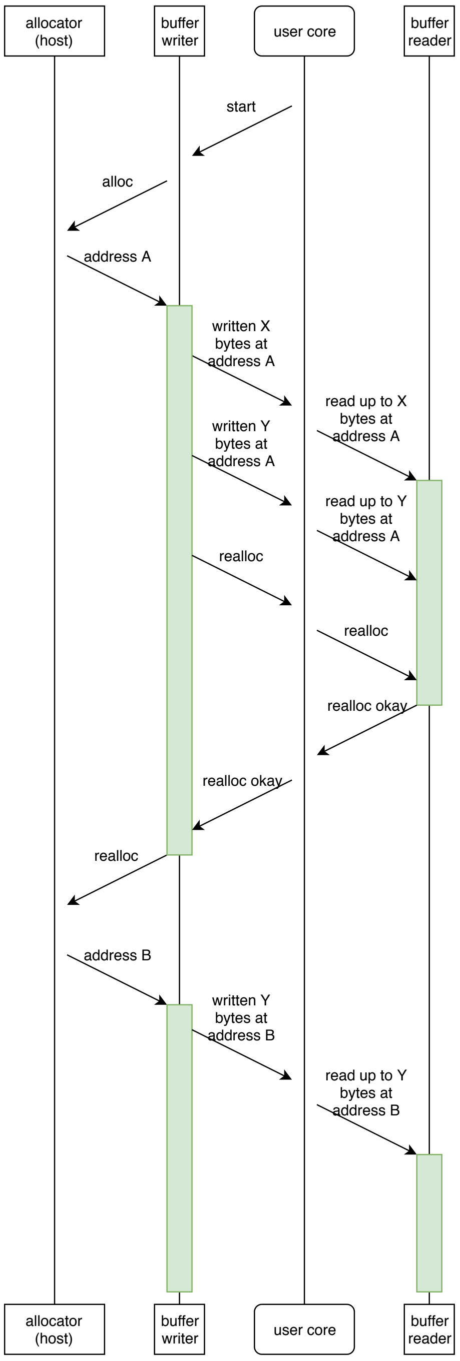

allocator (host)
buffer writer
user core
buffer reader
start
alloc
address A
written X bytes at address A
read up to X bytes at address A
written Y bytes at address A
read up to Y bytes at address A
realloc
realloc
realloc okay
realloc okay
realloc
address B
written Y bytes at address B
read up to Y bytes at address B
allocator (host)
buffer writer
user core
buffer reader

Figure 4.3: Concurrent buffer access when using realloc

## 4.6. Changes to Fletcher

The Fletcher wrapper generator will need to be modified to place all the new blocks at the right places. This kind of change is outside the scope of this thesis. In addition, it is necessary to provide the allocator with a write response channel, a channel that is currently non-existent in Fletcher. Since this change is critical to the correct functioning of the implementation, the write channel arbiter that is used in Fletcher needs to be augmented with a write response channel. This should be done in such a way that it is backwards compatible with the arbiter version that does not have a write response channel.

# 5 Implementation

The components are implemented using VHDL and should be synthesizable by tools from different vendors, although they were only tested with QuestaSim and Xilinx Vivado, targeting the AWS f1 systems. Functionality was added incrementally, so that there was always a working prototype with a subset of the required functionality. The component's interfaces are already fixed by the architecture, but this still leaves a lot of choices to be made for the individual components. The following sections start with details that affect multiple components, after which it describes the implementation of the individual components in more detail.

## 5.1. Virtual memory size

The virtual memory should at least be big enough to fit all the data that can be put on the FPGA, so at least the size of the FPGA board memory. However, the virtual memory can become fragmented to such an extend that large contiguous allocations are no longer possible. The largest possible contiguous allocation that can be satisfied at a given virtual memory usage is minimal when single-page allocations are spread uniformly over the virtual address space. For instance, with a virtual memory space of 16 GB and 3 pages allocated at a uniform distance, the largest contiguous allocation that can be made is slightly less than 4 GB. So when the maximum desired contiguous allocation, the maximum number of allocations, and the maximum total size of all allocations are known, the minimum size of the virtual memory can be determined to guarantee that these allocations will succeed.

When one takes the maximum number of allocations to be $n = 512$, the maximum total size of all allocations $T = 64\,\text{GiB}$, and the maximum size of each allocation to be $L = 8\,\text{GiB}$, the required virtual memory size $V$ is given by

$$V = L(n+1) + T = 4168\,\text{GiB}, \tag{5.1}$$

for which the next power of two is 8 TiB.

## 5.2. Page table organisation

The page size is a trade-off between performance and wasted memory. The larger the page size, the more memory is potentially wasted when allocation sizes are not a multiple of the page size, also known as internal fragmentation. Operating systems conventionally use a page size of 4 KiB for general purpose computing. Since Fletcher is usually employed in the big data domain, allocations will generally be large. Because of this, the percentage of wasted space will be low and page sizes can be larger than for general purpose applications, which to reduces the number of lookups and consequently improves performance. The maximum wasted space depends on the number of allocations and the page size. At most one page can be wasted for each allocation, so the maximum wasted space due to internal fragmentation with 512 allocations and a page size of 2 MiB is 1 GiB.

To keep lookup latency to a minimum and implementation complexity low, the number of page table levels should be low. Since a single-level page table would be large (256 MiB for 256 KiB pages and an 8 TiB virtual address space), a two-level page table is used in the implementation. For a two-level

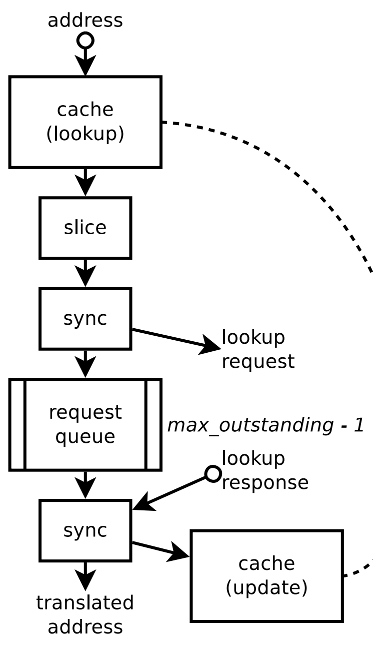


Figure 5.1: Structure of address translation unit.

page table and $\frac{8\,\text{TiB}}{256\,\text{KiB}} = 2^{25}$ pages, there should be $2^{13}$ entries per table in each of the two levels. With 64 bit entries, each page table would take up 64 KiB in memory.

## 5.3. Host communication

The host runtime software can send allocation and *free* requests to the FPGA trough the existing MMIO[1] interface. The FPGA acknowledges the command and provides a response on a separate MMIO register, which must be acknowledged by the host software. The allocation requests by the host and requests by other sources on the FPGA are multiplexed onto the same bus to the allocator. The existing host runtime library is altered to defer allocation and *free* operations to the FPGA allocator.

## 5.4. Address translator

The structure of the implemented address translator is shown in Figure 5.1. When a virtual address arrives at the input, first a lookup is performed in the local cache. Implementing an effective cache (TLB) is important [15], because misses are very expensive. The cache is fully associative, implemented in registers, and takes into account the validity mask that is produced by the page table walker. The cache can be updated by lookup responses in parallel to cache lookups. This is why the output of the cache lookup is caught in a register slice, keeping the lookup result stable for the next stage. The cache has a simple FIFO[2] replacement policy in this implementation and its size is limited to 32 entries due to timing constraints. A CAM[3] was considered for the cache implementation, but CAM is expensive to implement on an FPGA [16] [17].

The lookup result is stored in a request queue and if there was no entry found in the cache, a lookup request is send to a page table walker. Because of the request queue, this design can take advantage of a pipelined page table walker. At the output of the request queue, the responses of the page table walker are merged into the stream of addresses for addresses which were not translated by the cache. The sync blocks use StreamSync, the queue a StreamFIFO from vhlib [18].

## 5.5. Page table walker

The page table walker is pipelined, allowing it to perform multiple walks in parallel. Its structure is shown in Figure 5.2. Again, the sync blocks use StreamSync, the queues a StreamFIFO from vhlib.

[1]Memory-Mapped Input Output
[2]First In, First Out
[3]Content Addressable Memory

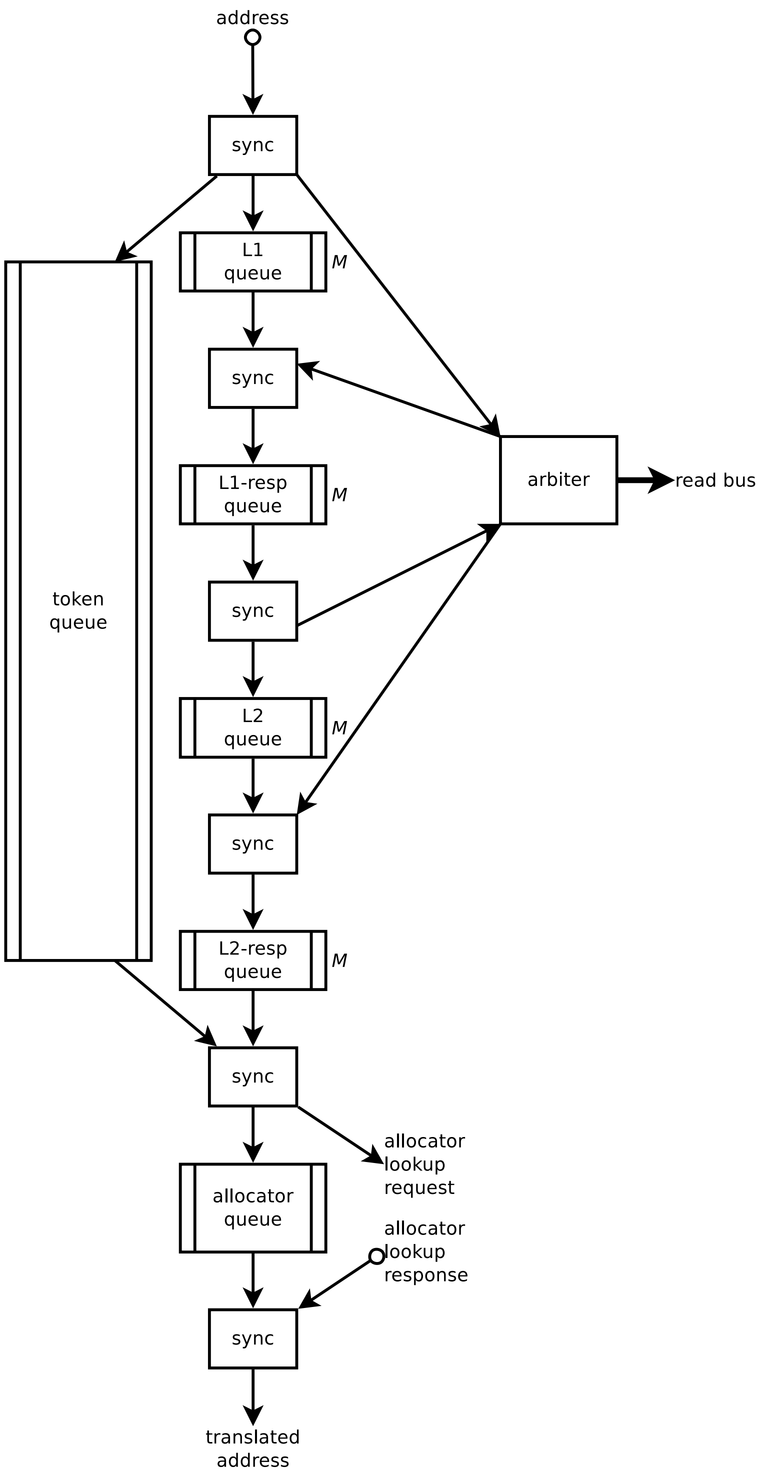


Figure 5.2: Structure of page table walker.

Each page table walk consists of two bus requests, one for the first level (L1), and one for the second level (L2). Each bus requests needs two queues, one to store information about the outstanding requests, and one to store the bus responses. Each response queue should be able to always absorb all outstanding requests. To prevent new requests from being generated when any of the queues could become full, a token system is used to limit the total number of outstanding requests.

If the page tables indicate that there was no physical memory allocated for a certain page, a request is sent to the allocator to provide physical memory for that page. The allocator will then respond with the corresponding physical address that will be used by the page table walker.

## 5.6. Allocator

The allocator is the most complex part to implement. There are also a lot of choices that need to be made, like what allocation policy will be used, where and how data is stored, what search algorithms to use. The first priority is to make a working allocator, not make a perfect one. So emphasis is on time to implement. The allocator will have two main tasks: allocate virtual memory, and allocate the associated physical memory. Of course this memory also needs to be freed eventually, but how this happens mainly follows from the decisions made for the allocation. To keep implementation simple, a first fit or next fit policy is chosen for virtual and physical memory allocation. Though these policies can increase external fragmentation [19], the virtual address space is sized large enough that fragmentation does cause issues, and external fragmentation is no issue for physical memory, since allocations can be split up into individual pages. A structural overview of the allocator is shown in Figure 5.3. The different components are explained in the next paragraphs.

### 5.6.1. State machine programming

A lot of internal action sequences are used at multiple points in different operations. To be able to re-use these, the state machine of the virtual allocator uses a stack to be able to execute these sometimes nested routines and then jump back to the state where these routines were "called" from. These routines are indicated by flattened circles in the state diagrams.

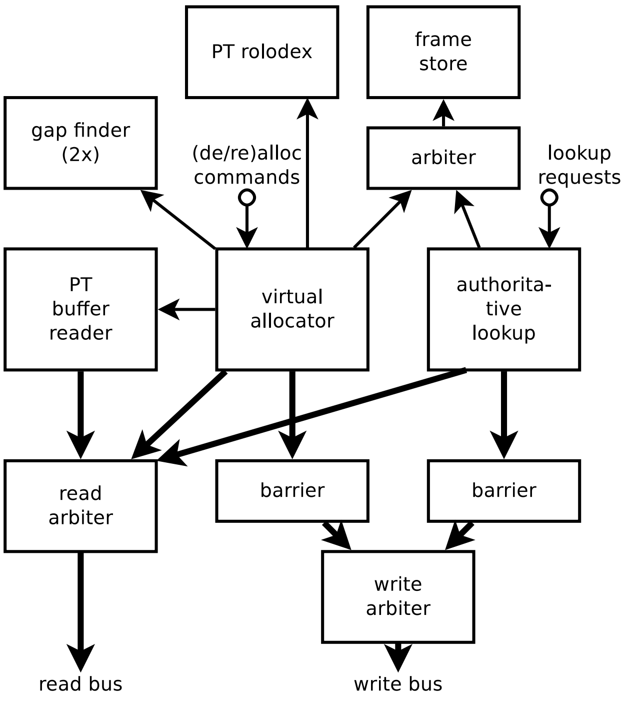


Figure 5.3: Structure of allocator and authoritative lookup unit.

### 5.6.2. Frame allocation

Virtual memory usage is stored in the page tables, these can be consulted when new virtual memory needs to be allocated. However, the usage of physical memory also needs to be tracked to be able allocate it. A lookup structure stored in on-board memory, like the page tables, is possible. However, since physical memory allocation may need to be done separately for each page on first access, searches for a free frame should return a result quickly. The features that would require a mapping from frames to pages, like evicting a range of frames [20], are not necessary in this implementation. Fortunately, only one bit for each frame needs to be stored and the size of physical memory accessible by the FPGA is limited to 64 GiB for AWS f1 systems, leading to a required storage size of 64 Kibit for 1 MiB pages. A table this small can easily be stored on the FPGA itself using BRAM, which has much lower latency than the on-board DRAM. The BRAM to keep track of physical memory memory is accompanied by logic to search it and encapsulated in the *Frame store* block. In this implementation, this BRAM is one bit wide, allowing only one frame to be examined or altered per clock cycle.

When searching for free virtual memory, the frame store makes use of a roving pointer for each memory region. A roving pointer stores the last location that was found to be free. Using a roving pointer prevents the frame store from scanning the (already occupied) lower end of the memory over and over for each search. The frame store can be accessed on different occasions: when allocating virtual and accompanying physical memory, when freeing virtual and accompanying physical memory, and when allocating a frame on a page's first access. This last action is initiated by the *Authoritative lookup* unit, which is separate from the allocator. Therefore, access to the frame store is arbitrated between the allocator and lookup unit. An initial address can be given to the frame store, which is currently ignored, but can be used to allocate adjacent frames to adjacent pages in order to leverage the address mask in the page table walker.

### 5.6.3. Packing page tables (rolodex)

Usually a page table is made to fit exactly in a page. Since our implementation is likely to use pages larger than 1 MiB, the page tables would be that big too, which would take a long time to initialize each time a new table is required. This, along with implementation restrictions from an earlier version of the allocator, lead to a solution where multiple page tables are packed into a single frame. Since both the frame and page table size are a power of two, an exact multiple of page tables fits into a single frame. The location where the first page table would be in the frame is reserved for a bitmap that indicates what locations in the page are currently in use. A complication of this is that when a new page table is needed, one needs to know which frames to search for free space. This role is filled by the *Page Table rolodex*, which gets its name from the fact that it flips through all frames that are used for page tables. One can insert new frames into the rolodex, or delete old ones. When the rolodex has flipped through all the frames in its memory, it signals that there are no more frames to examine, after which the allocator needs to allocate a new frame to create a new page table.

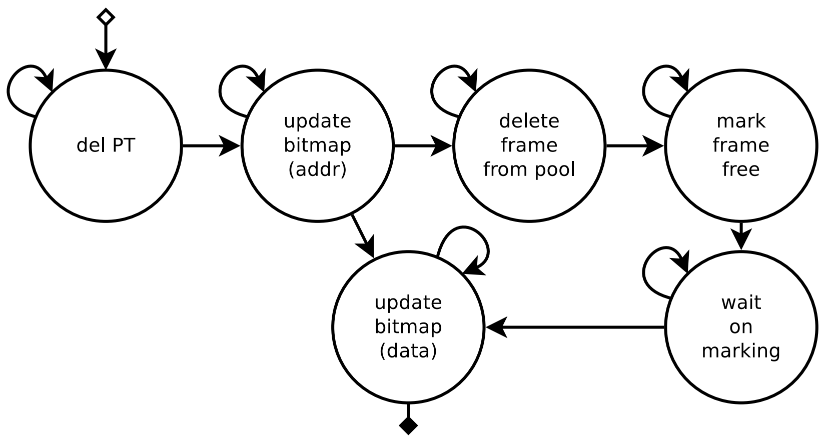


Figure 5.4: State diagram of page table deletion.

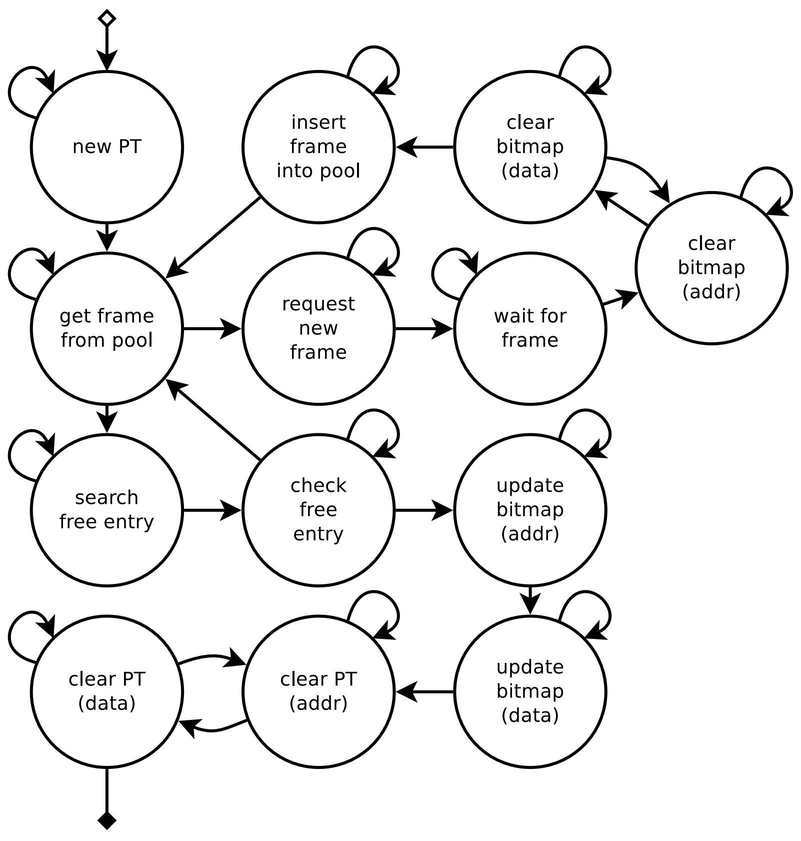


Figure 5.5: State diagram of page table creation.

Page tables are managed by the allocator, since the allocator is responsible for assigning virtual addresses. Due to the packing of multiple page tables per frame, the state machinery to manage page tables is a little involved. Steps to create a new page table are outlined in Figure 5.5, steps to discard a page table are outlined in Figure 5.4.

### 5.6.4. Authoritative lookup

The authoritative lookup unit is consulted when a page table walker cannot translate a certain virtual address. This can happen when a virtual address has not been allocated accompanying physical memory yet, or when the page table update that allocated it was not yet visible to the page table walker. Other causes include user error, which are not handled in this implementation. When the authoritative lookup unit determines that physical memory needs to be allocated, it requests it from the frame store. The frame store takes the memory region as argument. As soon as the frame store returns a frame number, the lookup unit forwards this to the requesting page table walker in order to minimize the reply time. After this, the corresponding page table is updated. This does mean that a lookup request for the same page can arrive to the lookup unit again. Either because a second page table walker requests the same page at the same time, or because the page table update is not yet visible to the page table walker. If the lookup unit were to do a page table lookup before its previous updates are visible to itself, it would allocate a different frame for the same page and overwrite the previous allocation. This is why the lookup unit must wait for a page table update to hit main memory, where it is visible to itself and the page table walkers. The barrier that is inserted after the lookup unit keeps track of writes that are not visible yet, by monitoring the write response channel. The lookup unit will hold off while there are still uncommitted writes.

### 5.6.5. Page allocation

Virtual address allocation is done by the virtual allocator. Since bookkeeping of the virtual address space is done in the page tables, the allocator is also responsible for managing the page tables. Due to project time constraints, the allocator only examines the first level page table for free space on

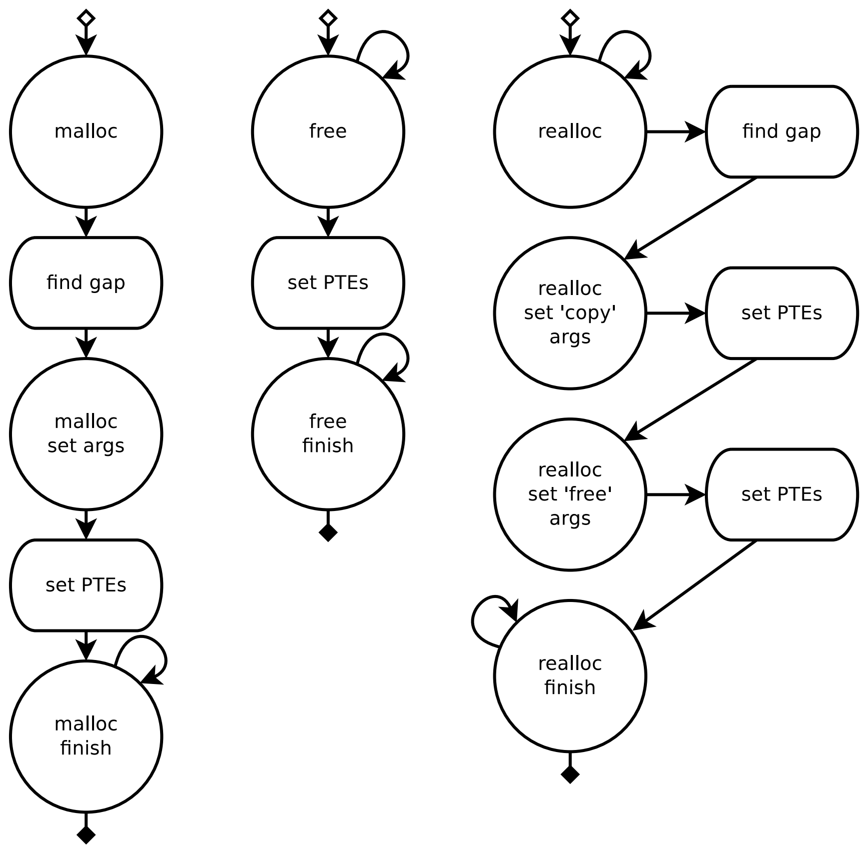


Figure 5.6: High level state diagrams of allocation, deallocation, and reallocation

allocation. Consequently, there will always be a whole first level entry (or second level page table) dedicated to a single allocation, limiting the number of allocations to the amount of entries in the first page table. The allocator will scan the first level page table until a large enough free space is found. After that, it updates the page tables, creating new page tables as needed. The memory region that the allocation is for is recorded in the leaf entries of the page tables and the last leaf entry of the allocation is marked, since the size of the allocation is not given when freeing it. Before the allocation address is reported back, the allocator makes sure the page table updates are visible to other components by waiting for the write response with the help of the barrier unit.

### 5.6.6. Deallocation and reallocation

When freeing an allocation, all the page table entries belonging to that allocation must be read in order to mark the associated frames as free. To know whether a page table is free after the freeing of the allocation, also the rest of each page table that is used by the allocation must be read. The page table can then be reclaimed for other allocations, or when a now unused page table was the last page table on the frame, the frame can be marked as free and used for other purposes. In order to speed up the scanning of the page tables, a buffer reader from Fletcher is used. The reader will read ahead and buffer responses from the bus, providing the deallocator with a semi-constant stream of page table entries.

The same system is used when reallocating. First, the page table entries are copied to a new location and more entries are appended. Thereafter, the page table entries of the old allocation are marked free. This process is illustrated in Figure 5.6. Much in the same way as when deallocating, except now the referenced frames are not marked free, because they are still in use ate the reallocated spot. All in all, when trying to reuse the same machinery for allocation, reallocation and deallocation, the state machine became quite complex, as shown in Figure 5.7.

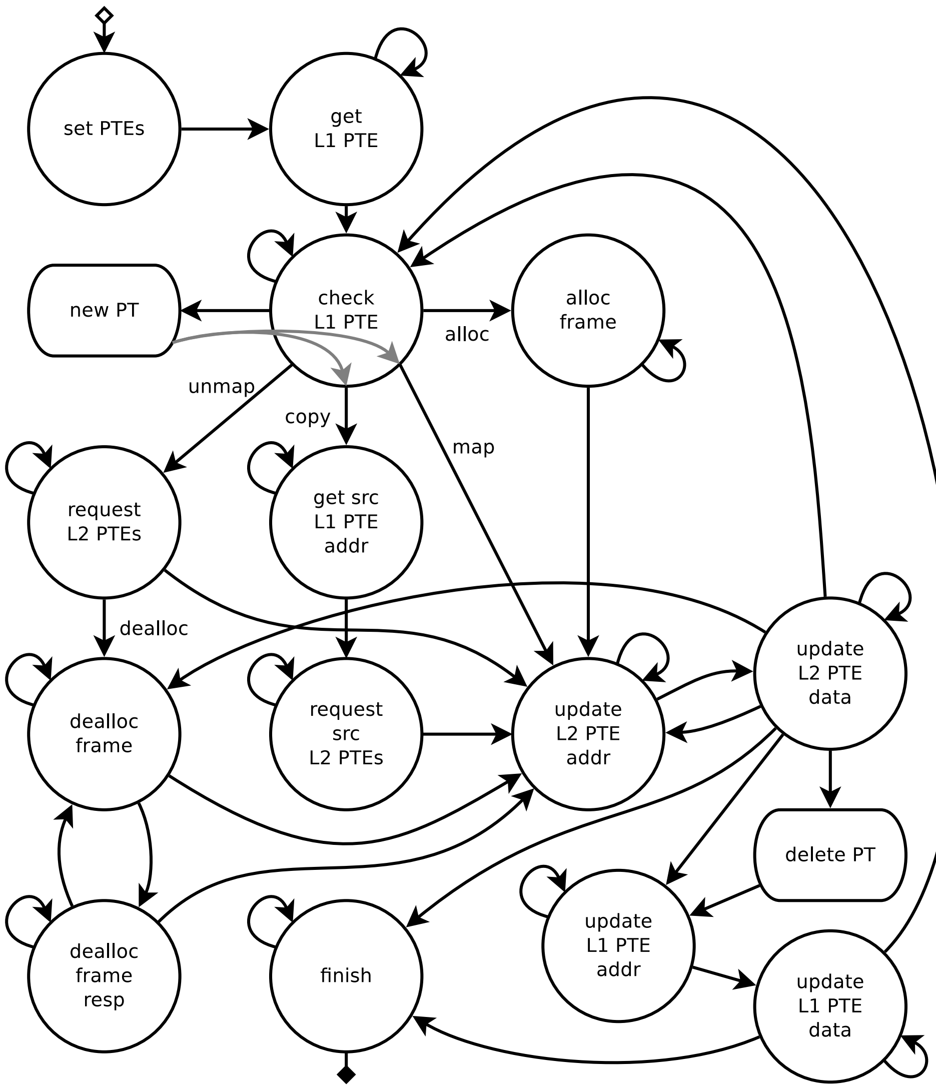


Figure 5.7: State diagram of page table entries updating.

A detail that needs to be taken into consideration, is that the lookup unit can allocate new frames in parallel. Even though an allocation should not be accessed after a *free* or realloc command is given, it could be that a page table update by the lookup unit is not yet visible by the allocator. To guard against this, the allocator waits until the lookup unit has had a response for each of its writes. A more elegant solution would be to implement an actual memory barrier on the read and write channels combined, but this was skipped due to project time constraints.

### 5.6.7. Initialization

When the system is first turned on, or reset, some steps must be performed to get it in a usable state. These steps are shown in Figure 5.8. First all frames must be marked unused. After that, the first level page table is created at a predefined location, which must align with the first possible page table in an arbitrary frame. The frame at this location is reserved for use for page tables, after which the entries in the first level page table are cleared. When this finishes, the allocator is ready to accept commands.

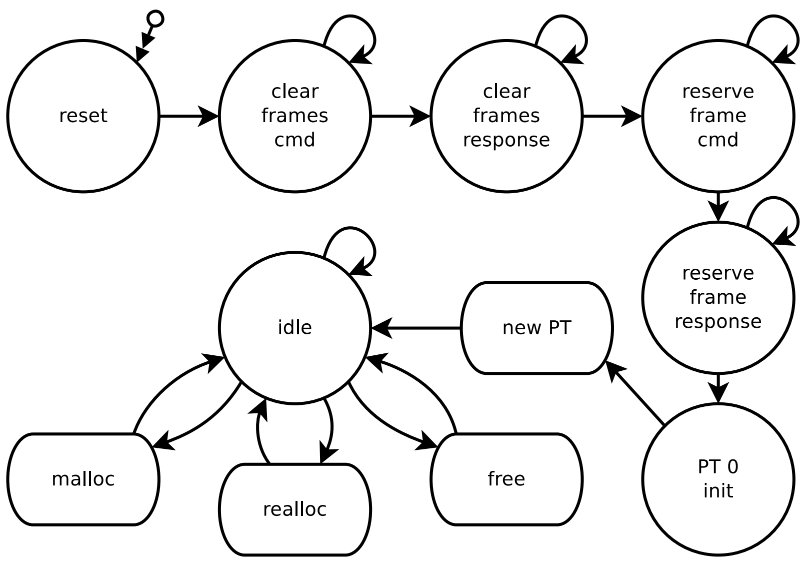


Figure 5.8: High level state diagram of allocator.

# 6 Measurements

The main question to be answered is: can the solution achieve near system bandwidth throughput? For this, we consider two cases: random access, and linear access. Any real application throughput will be somewhere between these two cases. An important parameter is the request (also burst) size, measured in *beats per burst*, where one beat equals 64 B in this implementation. Several implementation parameters like cache and page size are explored. Lastly, the performance of the allocation operations is examined. All of the measurements are done on an AWS f1.2xlarge FPGA instance, which includes a Xilinx UltraScale+ VU9P FPGA connected to four DDR4 memory channels, of which only one is used in these experiments. The design runs on 250 MHz, which is the highest bus speed offered by the AWS FPGA interface (shell). Unless otherwise noted, experiments use a page size of 64 MiB, one cache entry for the address translators, and a non-pipelined page table walker.

The baseline throughput to on-board DRAM can be found in Figure 6.1 and is near the maximum bus throughput of 16 GB/s. The memory interconnect that is used in the design has a limit on the number of requests per cycle (1/4), which limits the throughput for requests that take less than four cycles to complete. This is clearly visible in the throughput for linear access. The throughput for random access is lower due to the DRAM's internal processes. From this graph, one can derive that interleaved accesses for different data should use at least 16 beats per burst (1 KiB) to achieve maximum throughput.

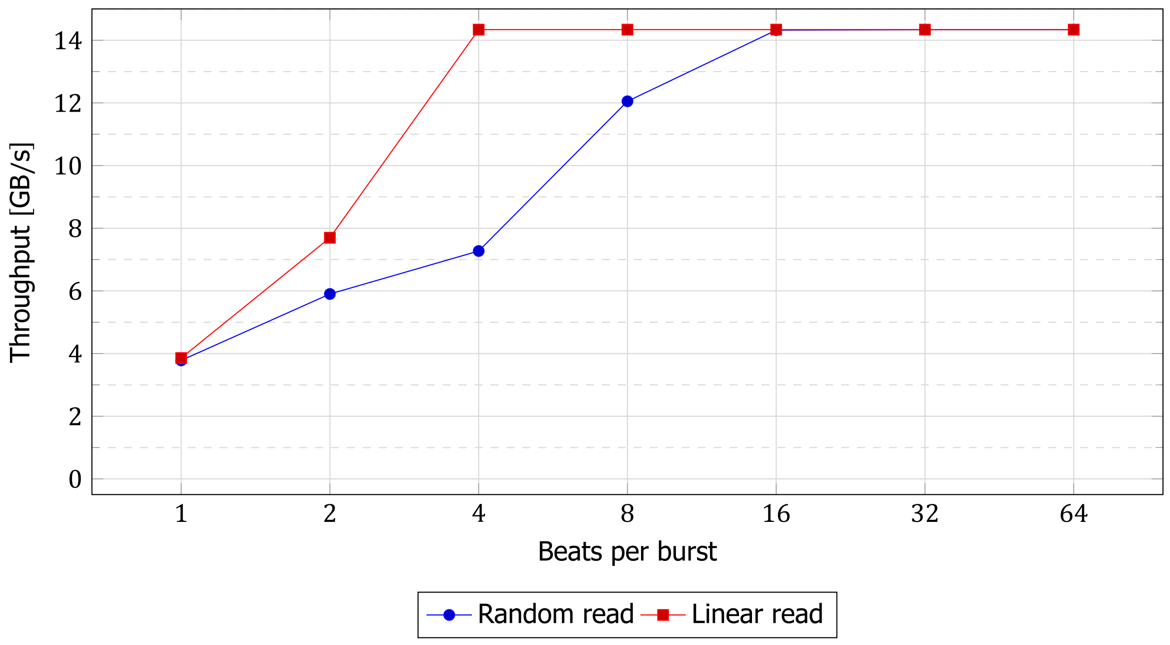


Figure 6.1: Device memory raw throughput (baseline)

An important aspect of the platform is the memory access time, or read latency. Since every (non-cached) virtual to physical address lookup requires two sequential memory accesses (one for each level of page tables), the latency for memory access through virtual addresses will about triple. In addition, if lookups are not pipelined, lookup requests need to wait on each other to complete while the memory bus sits idle. Random access read latency is $71 \pm 5$ cycles. Measured latency including virtual address translation is $233 \pm 23$ cycles, which is close to the theoretical triple latency.

## 6.1. Area

There exists a trade-off for FPGA area against performance for several implementation parameters like cache size and pipeline depth. However, the used area varies too much for small design changes to see any correlation. The area of the design, excluding the AXI interconnect is $5038 \pm 275$ CLBs or $3.4 \pm 0.2\,\%$ of total available area. A hierarchical breakdown of the area can be found in Table 6.1. The various arbiters are relatively big, mostly because they have additional registers on their in- and outputs to meet timing more easily.

The allocator contains a buffer reader, both in order to ease implementation and to speed up scanning of page tables. This turns out to be a significant portion of the design area. Replacing the reader with a thinner design would reduce area, but also increase the time it takes to complete *free* and reallocation operations, due to memory access latency. Fortunately, the allocator is a one-time cost and does not grow with the number of readers and writers. The number of translators and the area for some of the arbiters, however do grow with an increasing number of readers or writers.

| **design element** | **CLBs**[1] |
|---|---|
| f1 top | 5000 |
| translator(r) | 200 |
| translator(w) | 170 |
| translation arbiter | 190 |
| AXI top | 4600 |
| AXI MMIO | 350 |
| Fletcher wrapper | 4300 |
| bus read arbiter | 740 |
| page table walker | 540 |
| bus write arbiter | 440 |
| translation arbiter | 250 |
| reader translator | 140 |
| reader translator | 140 |
| random reader | 85 |
| linear reader | 70 |
| MMIO to allocator | 80 |
| allocator | 2600 |
| page table reader | 1200 |
| bus read arbiter | 760 |
| bus write arbiter | 730 |
| gap finder (page tables) | 280 |
| gap finder (virtual mem) | 50 |
| frame arbiter | 80 |
| frame usage bits | 15 |
| page table pointers | 35 |
| main allocator write barrier | 15 |
| lookup allocator write barrier | 5 |

Table 6.1: Hierarchical breakdown of FPGA area usage

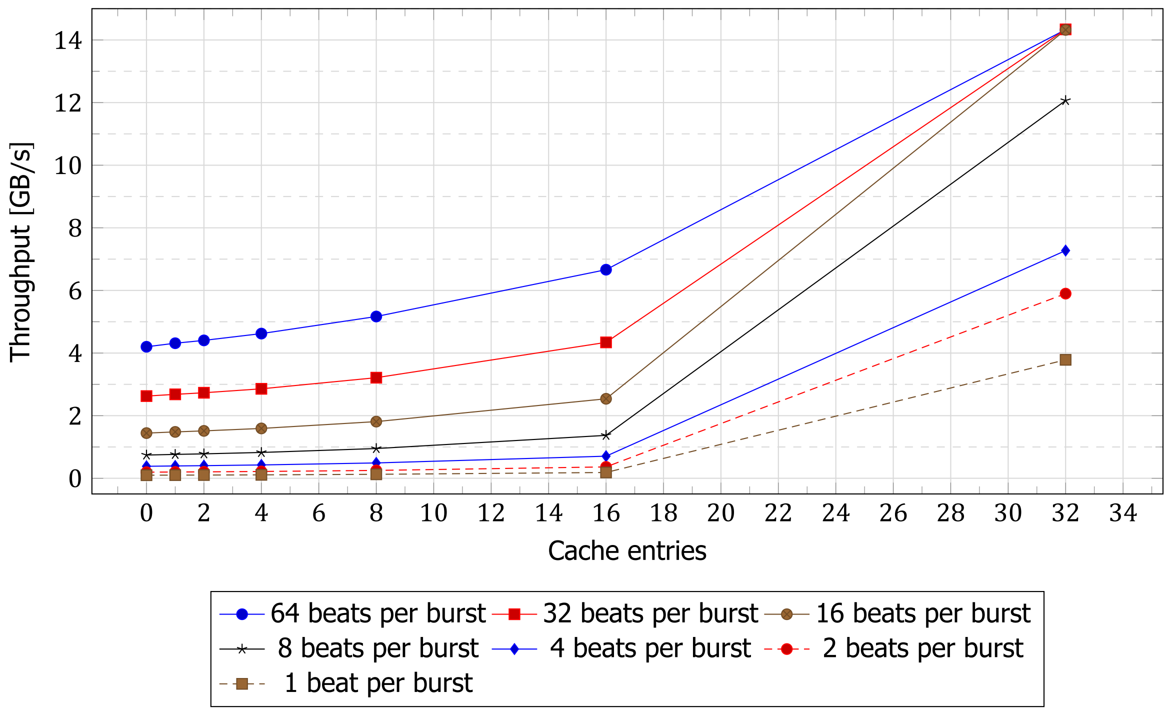


Figure 6.2: Device memory translated read throughput as function of cache entries, random access

## 6.2. Impact of translator cache size

For linear access, subsequent accesses will be within the same page most of the time. For this access pattern only one cache entry is required, and more is superfluous. Random access, however, requires more cache entries to enable high throughput. Each cache miss will require two sequential accesses to memory while the request for the actual data waits, making misses very expensive. To measure the effect of a cache, 0.5 GiB of data was read from a 2 GiB buffer using a random access pattern. The resulting throughput is plotted in Figure 6.2. Since the page size is 64 GiB, with 32 entries all address translations within the 2 GiB range can be cached and the throughput becomes identical to the raw throughput from Figure 6.1. Performance diminishes fast for smaller caches, especially for small request sizes.

## 6.3. Impact of page size

Although a larger page size makes the translator caches able to cache a larger address range, the page size does not have a direct influence on random access throughput. Therefore only linear access is measured in relation to page size, since there page size has a direct relation to the number of page lookups for a given amount of transferred data. In Figure 6.3 one can see that pages of 1 MiB and smaller start to have a noticeable impact on the system throughput. For larger pages the throughput is near the system limit.

## 6.4. Impact of page walker pipeline depth

While the page table walker is waiting for a response to a page table lookup, it could already issue a request for the next lookup. This can help increase throughput in cases where multiple lookups are required back to back, like non-cached random accesses. In this experiment, random accesses are made within a region of 2 GiB until 0.5 GiB has been read. Since the requested addresses are random and there is only one cache entry, there is a 97 % miss rate, which leads to an address lookup request for almost every address. The results for different amounts of outstanding lookups are plotted in Figure 6.4. For the non-pipelined version (single queue slot), the results are identical to the single cache entry throughput from Figure 6.2. Unsurprisingly, the throughput increases with an increasing number of queue slots, except for more than 21 queue slots, where there is a slight drop in throughput.

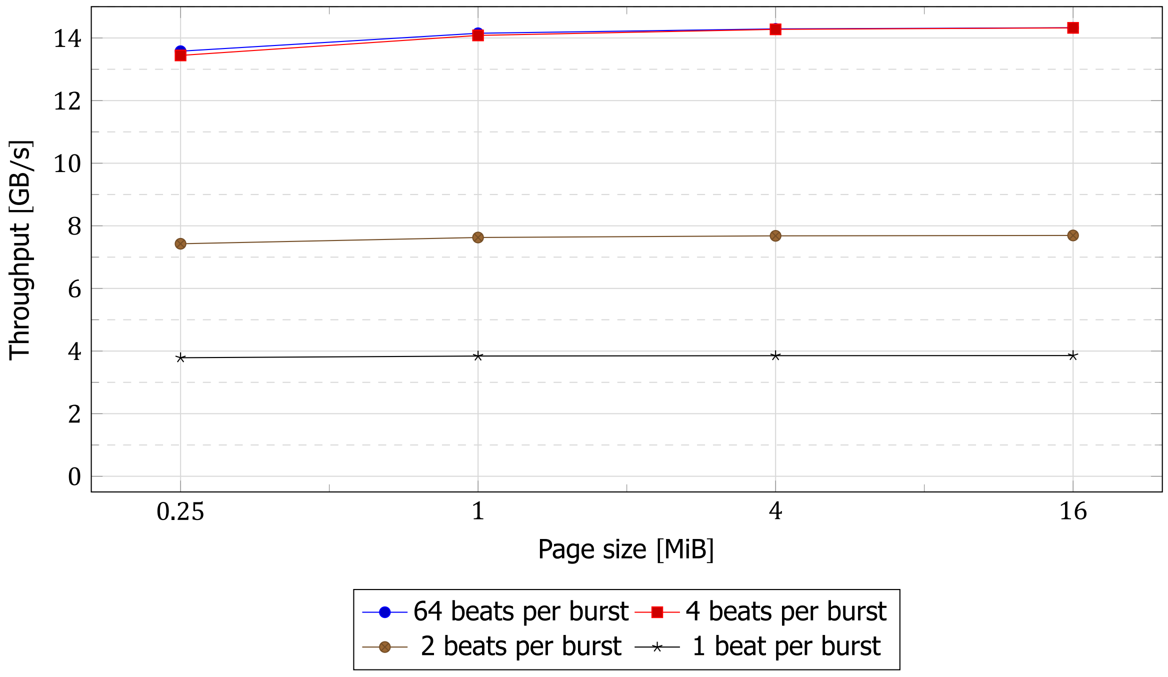


Figure 6.3: Device memory translated read throughput as function of page size, linear access

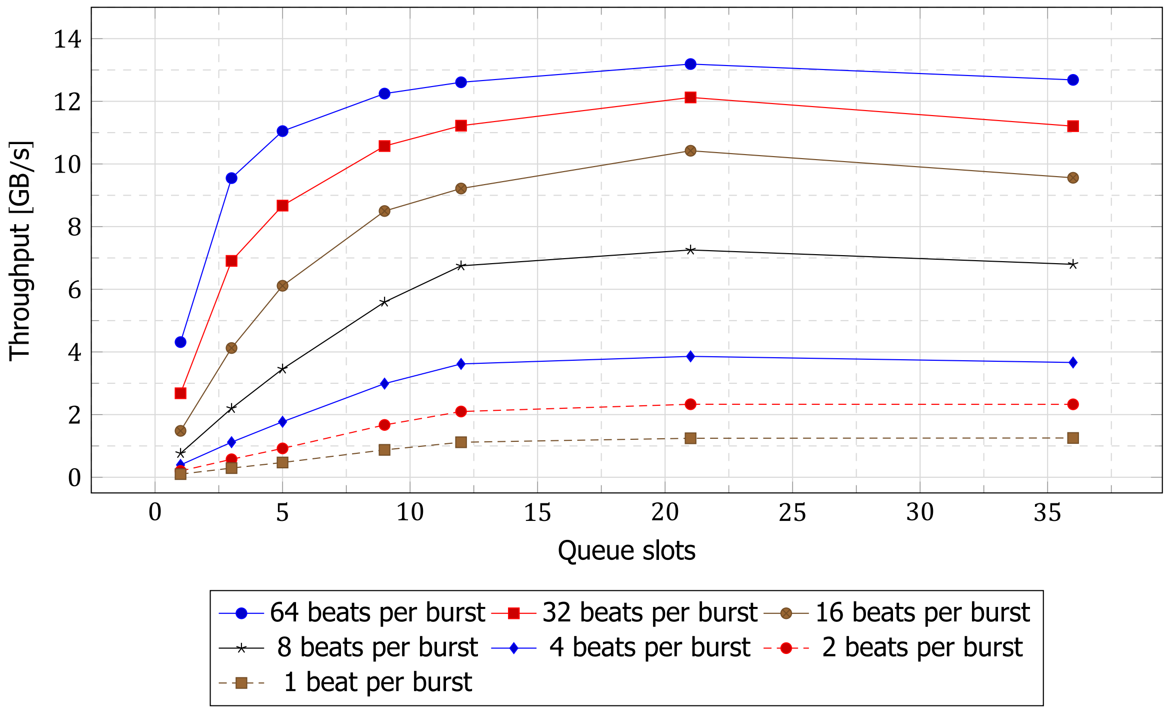


Figure 6.4: Device memory translated read throughput as function of queue slots, random access

The cause of the drop is yet undetermined.

## 6.5. Performance of operations

The time it takes for the various operations (*alloc*, *realloc*, *free*) to complete is relevant, though exact limits depend on the application. *Malloc* operations mainly impact an application's latency, while *realloc* can have an impact on the application's throughput, since it is used while writing out data, causing the writes to stall for the duration of the reallocation.

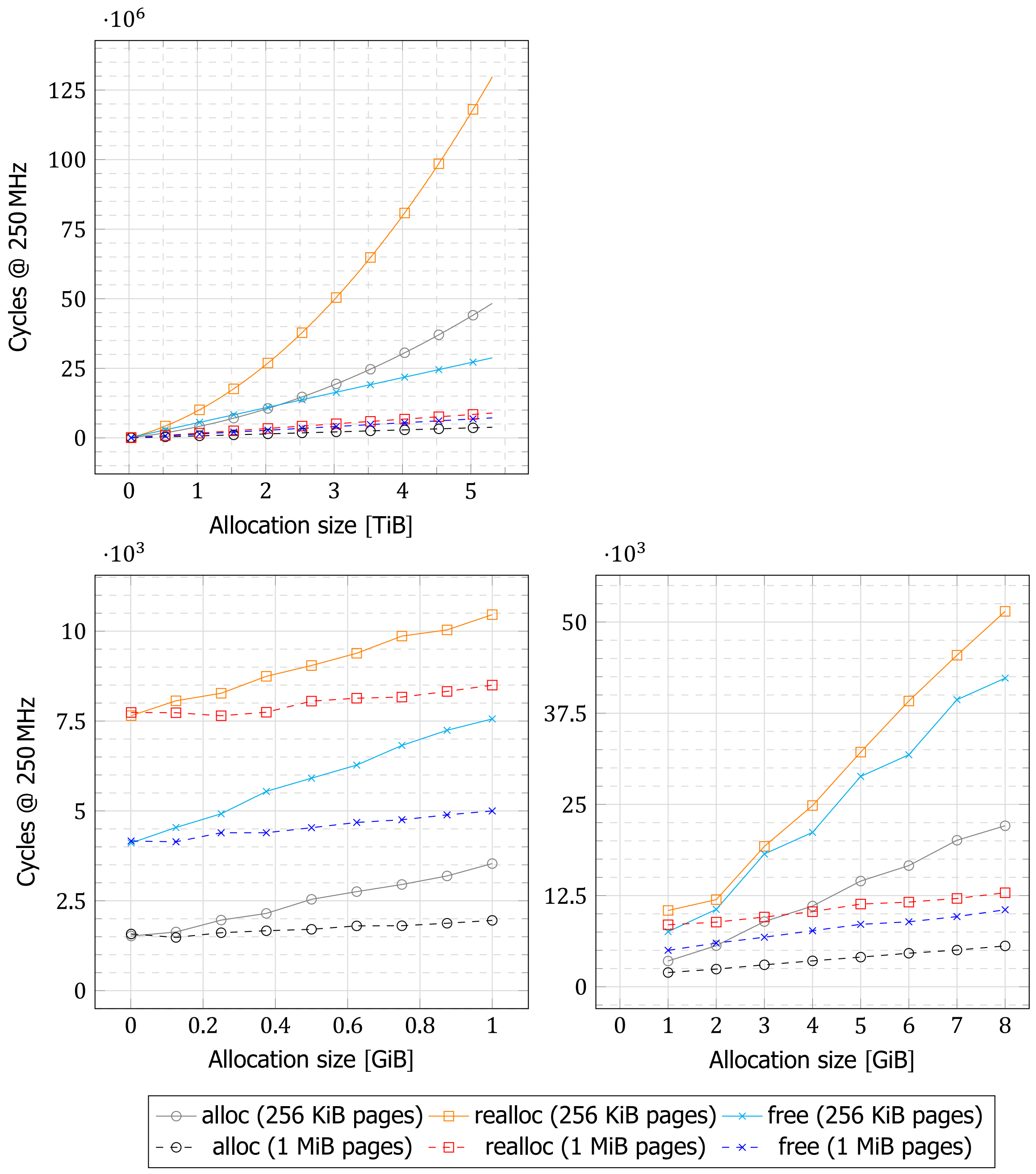


Figure 6.5: Operation latency for malloc, realloc, and free

The malloc and *free* times were measured alternatingly, always starting a malloc from an empty virtual space. The reallocation time measurements were done from a slightly smaller buffer, to the size indicated in the graph's axis. Fletcher uses Apache Arrow tables as data source and destination, which are recommended to stay limited in size to at most several gigabytes. However, to test the scalability of the virtual memory system, tests were also run for very large allocations. In Figure 6.5 one can see the exponential time it takes for alloc (and consequentially realloc) to complete. This is because the implementation needs to scan the bitmaps of all frames containing page tables in order to find a location for a new page table. The effect is much less pronounced when less pages need to be scanned, so when a lot of page tables fit within a single page.

Much of the cost for new small allocations is writing out a new page table, which takes over a thousand cycles for a 64 KiB page table. In turn, when an allocation is freed, that same page table must be read entirely to check whether it is currently unused and can be removed, in addition to marking referenced physical memory as free. A realloc operation is essentially the sum of an alloc and *free* operation, with the addition that there is already virtual memory in use at the time of the internal allocation. The size of the page tables could be made smaller for larger page sizes, lowering these costs as well.

# 7 Conclusions and recommendations

## 7.1. Conclusions

The first research questions was: Which memory management methods can provide the required functionality? Where the required functionality consts of: (a) allocate memory for buffers, (b) reuse memory from buffers that are no longer in use, (c) allow allocations to grow as required, (d) allow usage of multiple separate memories, (e) allow usage of host memory for platforms that support it, (f) present a malloc-like interface, (g) allow random access into the buffers.
We have seen several methods that can meet all these functional requirements. Among which were heap based memory management, segmented memory, and paged memory.

Which memory management method is the most appropriate, taking into account (a) performance impact, (b) memory usage overhead, (c) implementation complexity?
Paged memory is projected to have the lowest impact on write performance, have low internal fragmentation, and no issues with external fragmentation. Simple allocation algorithms can already be very effective for paged memory, making implementation relatively straight-forward compared to the other methods.

Can a suitable memory management method be implemented within Fletcher?
A general architecture was designed to manage paged memory, and an implementation made. The implementation was tested and evaluated on an AWS f1 system. The implemented system meets almost all functional requirements. It has a malloc-like interface, with the ability to allocate, grow, and deallocate buffers. Memory for old buffers is reused, and random access to buffers is possible. Additionally, it provides an easy to use "allocate on demand" feature, similar to the *overcommit* feature some operating systems offer. This works around some of the limitations that the reallocate function would impose on the system, like synchronisation between multiple units that access the same buffer when it is reallocated, at the cost of increased latency when accessing a page for the first time. The only functional limitation is that the translator's cache needs to be reset externally when switching buffers. This is to prevent stale entries from being used when a new allocation uses the same virtual addresses as a previously cached allocation. Ideally, addresses that become invalid by e.g. deallocation, are broadcasted to the translators so that they can invalidate the affected cache entries. Host memory allocation was not implemented, because it is not useful for the targeted platform.

What are optimal parameters for the implemented memory management method?
For a linear memory access pattern, a page size of 4 MiB or larger will have a negligible impact on performance (over 99.7 % of baseline throughput). For a random access pattern, performance is heavily dependent on the amount of data per request and can range from 30 % to 90 % of the baseline throughput when a pipelined page table walker is used with a pipeline depth of 21. To achieve good performance for a non-linear access pattern, it is important that effective translation caches are used.

What is the cost of the implemented memory management method in terms of

(a) performance
When a cache is employed that is effective for the access pattern of the application, the throughput can be over 99.9 % of the baseline throughput and access time is increased by only a few cycles (about 5 % on top of 72 cycles) compared to a system without virtual memory support.

(b) memory usage overhead
Paged memory suffers from internal fragmentation that leads to an average projected memory usage overhead of 0.2 % to 7 %. In addition, paged memory needs at least 64 bit per allocated page. The implementation, however, uses at least one (64 KiB) page table per allocated buffer.

(c) FPGA area
The implementation uses about 3 % of the CLBs available on the FPGA. Buffer readers will be less than 10 % larger due to the added virtual to physical address translation.

## 7.2. Recommendations

The AXI-like interface that Fletcher uses internally does not provide a transaction ID, like AXI has. Transactions are solely identified by the order they arrive in. As a consequence, transactions cannot be reordered and fast transactions need to wait on slow transactions. This makes it difficult to, for instance, make a cache on a memory bus work effectively. Since a transaction that is served by the cache cannot be finished before all transactions that were issued before it are finished, the cache would not be as effective. If transaction IDs are implemented in Fletcher, there would be more opportunities to speed up page table lookups by leveraging caches.

The implemented virtual memory allocation algorithm should be improved, so that one allocation does not take a whole page table for itself. Also, the page table walker does not implement aggregating multiple virtual to physical mappings when possible. When implemented, this will make caches in the address translators much more effective. To make this work, physical address allocation should be augmented to allocate consecutive frames with proper alignment.

To increase performance, the first level page table can be stored in BRAM. This is an alternative to using a cache for page table accesses. It would cut address lookup times almost in half, since only one of the two required memory reads need to go to relatively high latency DRAM.

If the memory overhead is acceptable, a single level page table may be interesting to implement. The implementation would be a lot simpler. In addition there would be the advantage of needing only a single lookup in DRAM, instead of the two lookups for a two-level page table for an address translation.

As proposed by Dirks [21], freeing of frames can be deferred to a later point in time, allowing free operations to return quickly and complete in the background later. Additionally, a list of free addresses can be maintained to speed up allocation under high memory utilisation.

Using speculative address translation may improve linear access performance for the smaller page sizes [15].

# Bibliography

[1] Apache Software Foundation, *Apache Arrow,* (2019).

[2] J. Peltenburg *et al.*, *Fletcher: A framework to integrate FPGA accelerators with Apache Arrow,* (2019).

[3] A. Bohra and E. Gabber, *Are mallocs free of fragmentation?* in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001).

[4] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, *Dynamic storage allocation: a survey and critical review,* in *Memory Management* (Springer Berlin Heidelberg, 1995) pp. 1–116.

[5] M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova, *Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures,* in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015* (ACM Press, 2015).

[6] E. D. Berger, B. G. Zorn, and K. S. McKinley, *Composing high-performance memory allocators,* ACM SIGPLAN Notices **36**, 114 (2001).

[7] P. J. Denning, *The working set model for program behavior,* in *Proceedings of the ACM symposium on Operating System Principles - SOSP '67* (ACM Press, 1967).

[8] OpenCAPI Consortium, *OpenCAPI consortium: Official site,* (2019).

[9] M. Adler, *Intel CCI: Core cache interface,* (2017).

[10] O. Chedru, *Memory management system for reducing memory fragmentation,* (2011).

[11] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, *Efficient virtual memory for big memory servers,* in *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13* (ACM Press, 2013).

[12] P. J. Denning, *Virtual memory,* ACM Computing Surveys **2**, 153 (1970).

[13] J. Evans, *A scalable concurrent malloc(3) implementation for freebsd,* (2006).

[14] M. Gorman, *Understanding the Linux Virtual Memory Manager* (PRENTICE HALL, 2004).

[15] S. Mittal, *A survey of techniques for architecting TLBs,* Concurrency and Computation: Practice and Experience **29**, e4061 (2016).

[16] X.-T. Nguyen, T.-T. Hoang, H.-T. Nguyen, K. Inoue, and C.-K. Pham, *An efficient I/O architecture for RAM-based content-addressable memory on FPGA,* http://arxiv.org/abs/1804.02330v3 .

[17] Z. Ullah, M. K. Jaiswal, Y. Chan, and R. C. Cheung, *FPGA implementation of SRAM-based ternary content addressable memory,* in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (IEEE, 2012).

[18] J. van Straten *et al.*, *vhlib: a vendor-agnostic VHDL IP library,* (2019).

[19] M. S. Johnstone and P. R. Wilson, *The memory fragmentation problem: solved?* in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98 (ACM, New York, NY, USA, 1998) pp. 26–36.

[20] R. van Riel, *Towards an O(1) VM: making Linux virtual memory management scale towards large amounts of physical memory,* in *Proceedings of the Linux Symposium* (2003) pp. 387–392.

[21] P. W. P. Dirks, *Method for allocation of address space in a virtual memory system,* (2000).