MASTER THESIS

Software caching for tree-based algorithms on accelerator cards

Peter Knoben

Faculty of Electrical Engineering, Mathematics & Computer Science
Department of Computer Architecture for Embedded Systems

EXAMINATION COMMITTEE
dr. ir. S.H. Gerez
dr. ir. N. Alachiotis
dr. ir. V. Zaytsev

June 2021

UNIVERSITY OF TWENTE.

Acknowledgements

This research would not have been possible without the help of others, whom I wish to thank. First of all many thanks to Dr. Ir. Nikolaos Alachiotis for being the main supervisor of this project. Because of the global pandemic we have only been able to meet a few times in person to discuss the the assignment and the progress. Despite that he was always there to guide me through the difficult times in the process and this research would have definitely not been possible without his help. I would also like to thank Dr. Ir. Sabih Gerez and Dr. Ir. Vadim Zaytsev for being part of the graduation committee and providing feedback on the proposal as well as the final report. Lastly I would like to thank Eline and Sjoerd for keeping me motivated and focus during the difficult times I faced during the process.

Abstract

Modern hardware accelerator cards create an accessible platform for developers to reduce execution times for computationally expensive algorithms. A powerful computing system can be created by combining a hardware accelerator and a processor. In this system the processor facilitates the overhead control of an algorithm and the hardware accelerator provides computational power for the intensive calculation in the algorithm. Such a hybrid system also has a downside. Most widely used systems have dedicated memory spaces, resulting in the processor having to transfer data to the accelerator card memory space before the computation can be executed. Currently the performance increase from using a accelerator card for data-intensive algorithms is limited by the data movement, this is called the *memory bottleneck*.

This research aims to reduce the effect of this memory bottleneck and improve overall performance by caching data on the accelerator memory. Caching is a proven technique in other fields as a method to reduce data movements and shows potential for hardware accelerator cards in certain use cases. Caching exploits the locality of data to reduce data movements. Tree-based algorithms inherently provide this locality of data due to the structure of the tree.

The goal is to design a software based cache running on the host processor to utilize the accelerator card memory space as cache memory to verify that caching can alleviate the effect of the memory bottleneck. The designed cache is tested on the Phylogenetic maximum likelihood function, a data intensive tree-based algorithms to calculate evolutionary relations between different species based on genetic characteristics. This algorithm has shown to be susceptive to the memory bottleneck.

The tests showed that in the best scenarios the number of data movements from host to accelerator card memory for the phylogenetic likelihood function is reduced by 90%. This provides a reduction in the total execution time between 31.6% and 39.9% for different tree sizes. Given the results caching on hardware accelerator cards shows potential in reducing the memory bottleneck for tree-based algorithms and is worth further investigating on other algorithms.

Contents

List of acronyms

AAT	Average Access Time
ADR	Action Design Research
AMAT	Average Memory Access Time
API	Application Programming Interface
AR	Action Research
ASIC	Applications Specific Integrated Circuits
AWS	Amazon Web Services
CPU	Central Processing Unit
DNN	Deep Neural Network
DSRM	Design Science Research Method
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GBDT	Gradient Boosted Decision Tree
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Units
HA	Hardware Accelerator
HDL	Hardware Description Language
HLS	High Level Synthesis
IS	Information Systems
LFU	Least Frequently Used
LRU	Least Recently Used
LSB	Least Significant Bit
MFU	Most Frequently Used
MRU	Most Recently Used
MSB	Most Significant Bit
PCIe	Peripheral Component Interconnect Express
PLF	Phylogenetic maximum Likelihood Function
RP	Replacement Policy
RTL	Register Transaction Layer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

Combining CPU processors with hardware accelerators creates incredible new possibilities for high performance computing. It also creates some new challenges along the way as explicit data movement is required between the host processor and accelerator cards in the absence of shared memory space.
A way to possibly reduce the effect of this bottleneck is caching. Caching exploits the locality of data to optimize the data transfers. In tree-based algorithms the relation between data is known, allowing to be exploited by caching.
That is why research aimed at exploring the effectiveness of software caching for tree-based algorithms when using accelerator cards is proposed in this report.

1.1 Motivation

Hardware accelerators are being used more and more to reduce computation times for time-intensive algorithms. However hardware accelerators do not always share a memory space with the host processor. Examples of this are data center architectures or accelerator cards. When accelerator and host do not share a memory space, as shown in fig. 1.1.1, the data has to be transferred to the accelerator dedicated memory.

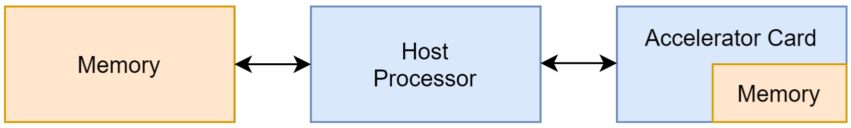


Figure 1.1.1: Block diagram of memory architecture for host processor with accelerator card.

In order to do so first the host processor must retrieve the data from memory. It is then transferred to the accelerator card and stored in the card's memory. The data can now be used by the accelerator card and the result is stored in the same on board memory. Finally the host processor can retrieve the data from the accelerator card memory space and transfer it to the host memory.

Data transfers in memory-bound applications are time expensive [1]. When extra time is required for data transfers the performance improvement perceived by the host is reduced. This is called the *memory bottleneck*. The effect of the memory bottleneck is especially noticeable in data-intensive algorithms and can lead up to a point where the use of a hardware accelerator has no positive effect on the performance anymore. For data-intensive algorithms performance improvement of a hardware accelerator is limited by the data transfer speed between both memory spaces. This is a general problem for hardware accelerators [2] and is sometimes also referred to as the I/O bottleneck.

To reduce the effect of this memory bottleneck multiple possible improvements can be made, with the two most conspicuous improvements being, faster data transfers between both memories or reduction of the amount of data transfers. The transfer speed between a host and accelerator card lies outside the scope of this research. The focus will be on reducing the amount of data transfers in data-intensive memory-bound applications.

This problem is not new. In the early days of processors reading and writing to the memory quickly became time expensive [3]. To overcome this memory bottleneck caches were invented. A cache is a memory space very close to the processor, which reduces the time spend on read and write operations to that memory with respect to read and write operations to other memories available/accessible [4]. More research is required to see if the behavior of a cache memory can be mimicked in software for an accelerator card to reduce the memory bottleneck effect by using on card available memory space as cache.

Research problem

Realized performance of hardware accelerators with dedicated memory is reduced when explicit data transfers to and from accelerator-dedicated memory are required.

1.2 Scientific contribution

Caching is an established and well-used concept in processor design. All modern processor cores have a form of cache memory. This research will implement this core concept in computer architecture and apply it in a novel way. The existing memory hierarchy control of a system is extended to the memory space of an accelerator card. This allows the memory on an accelerator not only to be used as a buffer for a compute kernel but to be used as a genuine memory space where data can temporarily be stored.

With this change in memory usage on the accelerator card, the goal is to reduce data transfers between host and accelerator card for data-driven algorithms to alleviate the memory bottleneck that is currently present. Standard caching techniques, currently used for processors, will be evaluated on tree-based algorithms as these algorithms inherently have locality of data that can be exploited by caching. The different characteristic of caching are evaluated separately in this research, to show how it influences the performance when implemented on an accelerator card.

Finally the entire caching design is tested on the phylogenetic maximum likelihood function algorithm as proof that caching can alleviate the effect of the memory bottleneck in real world applications.

1.3 Thesis structure

The structure of the thesis is as follows. In the next chapter 2 the background of the project is explained. Next in chapter 3 the objectives are listed and different possible research models are discussed. In chapter 4 existing literature is reviewed. Next the design of the cache is described in chapter 5. In chapter 6 the evaluation method is described followed by the results in chapter 7.The found results are discussed in chapter 8. Finally in chapter 9 the conclusions from this research are drawn.

Chapter 2

Background

In this chapter some background information about the project is given. Section 2.1 describes different hardware accelerators and how hybrid systems with accelerators can be made. Next section 2.2 introduces caching as a method to reduce these downsides. Lastly some background about tree-based algorithms as potential use case is given in section 2.4.

2.1 Hardware accelerators

The potential of hardware accelerators is great and well known nowadays. That is why in many applications hardware accelerators are used, in bioinformatics and computational biology [5], modeling molecular interactions [6], remote sensing and cryptanalysis [7] and general microprocessor implementations [8], [9]. Hardware accelerators can utilize parallelism by having a more concurrent nature than traditional sequential software counterparts. Therefore hardware accelerators are very suitable for speeding up computationally intensive applications.

The most commonly known hardware accelerator technologies available today are Field Programmable Gate Array (FPGA), Graphical Processing Unit (GPU) and General Purpose Computing on GPUs (GPGPU) and Applications Specific Integrated Circuits (ASIC). There are many more hardware accelerators but most are, like the ASIC, more application specific. FPGAs and GPUs are generic and reconfigurable or programmable and are therefore easy to use during development. Both options are available with PCI-express (PCIe) interfaces. This allows for high bandwidths, with PCIe 4.0 on x16 lanes up to 32GB/s, and easy interfacing from a CPU. The main vendors for FPGAs as well as GPUs all offer accelerator cards with PCIe interfacing.

The combination between the versatility of a processor and the computational power of a hardware accelerator can be the base of very useful hybrid systems, especially when a high bandwidth interface like the aforementioned PCIe is used.

2.2 Caching

One of the most commonly known techniques to reduce the effect of a memory bottleneck in a system is caching. A memory bottleneck essentially is a process throttled by a high latency. Caching can reduce this latency but can also impact the throughput of the process.

2.2.1 Latency

A cache is a memory space very close to the user of that data for temporary storage. By having the data closer to the actual user the transfer speed is much higher, thus reducing the memory bottleneck. A great and commonly known example is the cache on a CPU. This is a small memory space on the actual processor itself. The processor in this case being the user of the data. Here data can, temporarily, be stored so the next time data is required the processor does not have to transfer it from the main memory, reducing the number of transfers between the main memory and the processor. Memory space so close to the processor is very expensive and thus very limited in size. A hierarchy is thus created with different types of memory within a system. For hardware these hierarchical memory management systems often go even deeper with multiple levels (L1 to L3) within the cache memory.

Another relatable example are web browsers. Last visited websites are stored on the user side of the network, i.e. your computer. If the user wants to visit that website again the content does not have to be retrieved from the remote server but can be read from the web browsers cache memory, resulting in lower latency for the user.

2.2.2 Throughput

Besides the latency reduction caching can also improve the throughput rate from underlying resources. The overhead for data transfers is not linearly related to the data size. An intuitive example for this is the memory address. When a processor wants to retrieve data from memory it has to send an address from where to read the data, that takes up bandwidth. If the processor in that case only reads a small amount of data the ratio between transferring the address versus transferring the actual data is high. When these small grained data accesses can be combined to form one larger data transfer this ratio will lower, increasing the efficiency of the bandwidth. The extra small grained data fragments can then be stored in the cache for when that data is actually required.

Using combined data transfer sets, the costs of data transfer per data item are reduced and so is the memory bottleneck effect. In order to benefit from this higher grained data transfers, information might be required to know which data will be requested in the future. For this reason caches rely on the locality of data.

2.2.3 Locality of data

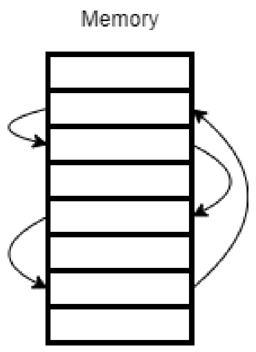
One of the principles that help the performance of a cache memory is the locality of data in the underlying resources. There are two types of locality that can be distinguished for cache performance improvements, temporal locality and spatial locality. Both will be explained below.

Temporal locality

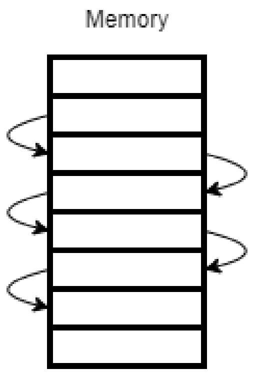
The temporal locality of data relates to the moment when the data is required. Often these moments are close together making it beneficial to store that data temporarily in cache. The website caching example, as mentioned before, by web browsers shows the concept of temporal locality of the data in the form of web pages. More specifically for this example, a search engine. When using a search engine the user is greeted with a list of possible websites containing the desired information. It is very likely that not all of those websites contain the actual information or that the user intents to compare different results. Every time the user wants to check a different result he/she returns to the web page of the search engine. Making it very likely that the user requires the information on that page multiple times in a short time frame, making it ideal information to store in cache memory. This is just an intuitive example. In a lot of processes data is often addressed based on the previous addressing moments. In fig. 2.2.1a a cyclic memory access pattern is shown. Here it is clear that recently used memory addresses are accessed again showing this temporal locality. In this figure the arrows represent the access order of the data.

Spatial locality

Besides the temporal aspect there is also the spatial aspect to locality of data. In the actual sense that data used together is often also stored together, in which case it might be beneficial to not only transfer the called data to the cache but also some surrounding data in a large grain data transfer. This way the throughput as described before is increased and future memory calls can possibly be prevented. Such spatial locality of data on a memory level is shown in fig. 2.2.1b.



(a) Temporal locality



(b) Spatial locality.

Figure 2.2.1: Access orders on memory level.

2.2.4 Integrated

One of the true beauties of caching is the added benefits for the user, without additional efforts. In the sense that it, in most cases, it is an integrated system operating on its own. A properly implemented cache does not require any interference from the user. So for the user nothing really changes but now the data transfer and thus latency is sometimes improved. The fact that almost all modern computing devices have a cache memory shows that there is benefit in this hierarchical memory for the user.

2.3 Cache architecture

Now that the working principles of the cache are clear architecture of a cache is elaborated upon. Just like a regular memory space, cache memory is divided into blocks. These blocks can have an arbitrary size but for the explanation each block, in both the main and the cache memory, contains one byte of data. The number of blocks in a cache is at least an order of magnitude smaller than the number of blocks in main memory. There are three main aspects important to understand how a cache operates.

- The determination where in cache data must be stored.
- The determination where in cache data can be retrieved.
- Which cacheline to replace when the cache is full.

2.3.1 Direct mapped configuration

First of is the determination of location in cache memory. Just like the main memory each block in cache is assigned an index. Now every address from main memory is mapped to an index in cache, this is called a direct mapped configuration. When the number of blocks in cache is a power of two, a simple mapping method can be used to determine the index in cache for any arbitrary main memory size. For example take the memory spaces in fig. 2.3.1. When the number of blocks in cache is in the form of 2^n, the controller can lay a mask of the last n bits over the address in main memory and map each address to an index in cache. Since the address will just roll over, the bit mask, a repeating pattern emerges and the mapping starts at the top of the cache again. Each cache index can be determined by $cache\_address = main\_address \% 2^n$.

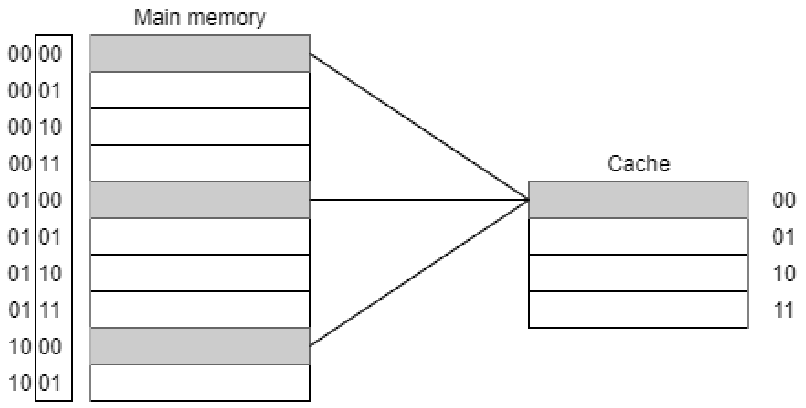


Figure 2.3.1: Direct mapped cache memory.

This way every index in cache is assigned an equal amount of main memory addresses. This is the simplest form and can all easily be realized in hardware with some logic without requiring the processor. This method does have its downsides. Since it is a fixed system, there is no flexibility for the controller to store data somewhere else in cache. If for example in a set of data, the data is alternated with metadata about each byte, which is not required by the processor, the cache memory space cannot be entirely utilized. This is shown in fig. 2.3.2. Here it can be seen only half the cache memory is addressed and the other half will never be utilized. The same effect could happen if the block size in main memory and in cache memory differ from size. If the block size in cache is twice the size as the block size in main memory, half the cache will never be addressed. Other configurations are possible to overcome this lack of flexibility, these are explained later on in section 2.3.2 and section 2.3.3. First the way the controller determines if data is already in cache, where, or that it should still be retrieved from main memory will be elaborated.

Determining if and where data is or needs to be is not a straightforward principle. Since multiple addresses of the main memory map to the same address in cache, it is not enough to check if there is data in a specific block in cache. The controller needs a way to check if that is the data the controller is looking for. In order to do that a tag is added in the cache memory to each data block. This tag can essentially be anything as long as it uniquely identifies a data block in main memory. In general the address in main memory is chosen as a tag. This is convenient since it is a unique identifier and furthermore the address is already required for the controller to locate the block in cache. When data is transferred to cache memory, the address is copied to the tag field of that block in cache. There still is a small optimization step that can be applied here. Since part of the address is already used to find the index in cache as mentioned before. Those bits are already known and do not have to be stored anymore. For instance with a cache of 64 blocks (2^6) and an address in main memory of 32 bits. In this case the mapping will look like something in fig. 2.3.3. Here only 26 bits of the address are stored as tag.

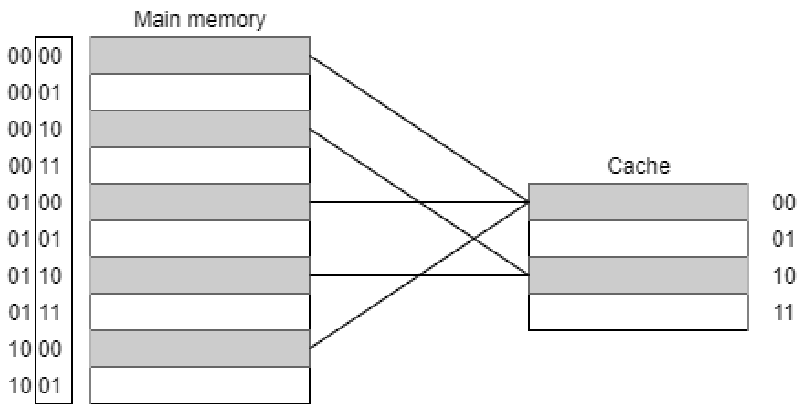


Figure 2.3.2: Alternating data pattern.

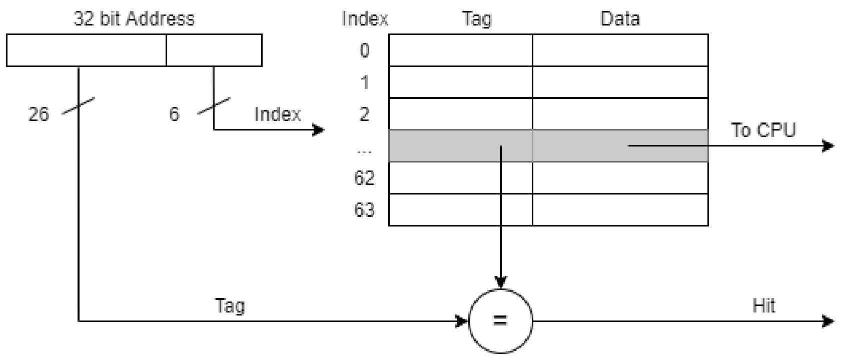


Figure 2.3.3: Mapping logic adapted from [10].

There is one final aspect still missing from the cache. When the cache is initialized there might still be data and tags in the cache which are not relevant anymore but the controller cannot make this distinction without extra information. So besides the tag field another field is added to each cache block in the form of a valid bit. This is just a single bit indicating if the data in that block is valid, indicated by a '1', or not valid, indicated by a '0'. This way the controller can determine if the data in the cache block is the data it is looking for or just random data that is still there from initialization. The logic added to the system can be seen in fig. 2.3.4. Here both the tag and the valid bit must be true for the data to be in cache.

An added benefit of this valid bit is that on initialization only the valid bits have to be set to zero. All other memory space does not have to be cleared or set to a known value. It will just

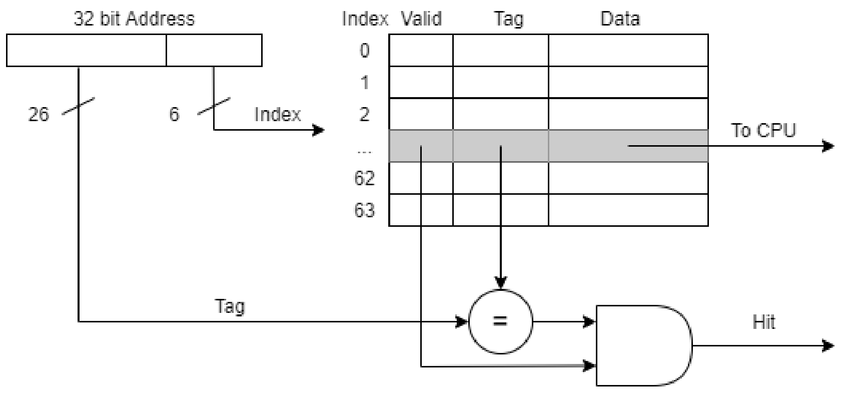


Figure 2.3.4: Mapping logic with valid bit adapted from [10].

be overwritten to useful data when something is written to cache. The combination of this data block, tag field and valid bit will be from hereon be referred to as a cacheline. The index of the cache will not only point to a data block but to an entire cacheline.

2.3.2 Fully associative configuration

As mentioned in the previous section, such a basic direct mapped cache architecture can have downsides. The lack of flexibility limits the cache performance in some instances. There are other architecture designs which do have this flexibility. One example is a fully associative cache. Here data can be stored in any cacheline instead of being mapped to a specific cacheline. This allows for full cache utilization for any data set. However, just like the direct mapped configuration, there are some downsides. First of all, to uniquely identify the data in cache the entire address needs to be stored as tag since the index field is no longer taken from the address. This also increases the logic required to check if the data is already in cache. The tag of each cacheline has to be compared to the new address. Not only does this add logic it adds complexity. Now the controller has to check every cacheline to locate data in cache. The logic required for this can be seen in fig. 2.3.5.

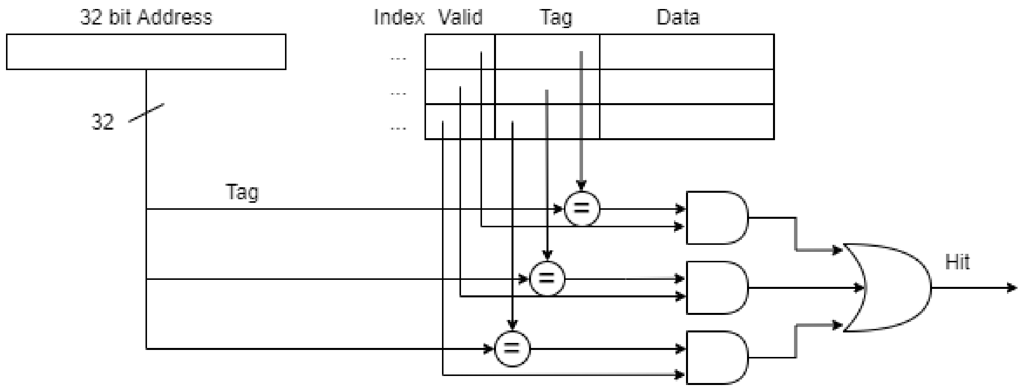


Figure 2.3.5: Fully associative cache memory logic adapted from [10].

There is however another difficulty with fully associative caches. When the cache is full the controller has to determine which data to overwrite with the new data. To determine the overwriting of data a controller requires a replacement policy. The replacement policies will be further elaborated upon in section 2.3.4 after the different cache configurations.

2.3.3 Set associative configuration

A direct mapped cache is simple, relatively cheap in terms of logic but not flexible. On the other hand a fully associative cache is highly flexible but more complex and requires more logic. There is another cache configuration type which combines best of both worlds and acts as a middle ground between the two; a set associative cache. A set associative cache can be seen as a direct mapped cache but instead of each index pointing to a single cacheline it points to a set of cachelines. Within this set there are multiple cachelines where the controller can store the data. Each cacheline in a set is called a way. If a set has 2^n blocks, or ways, it is a 2^n-way associative cache. In fig. 2.3.6 the different configurations for a cache with 8 blocks are shown. A larger set size results in a higher level of associativity within the cache. Note that a 1-way associative cache is just a direct mapped cache and the 8-way associative is a fully associative cache. The logic for a 2-way associative cache will look something like fig. 2.3.7

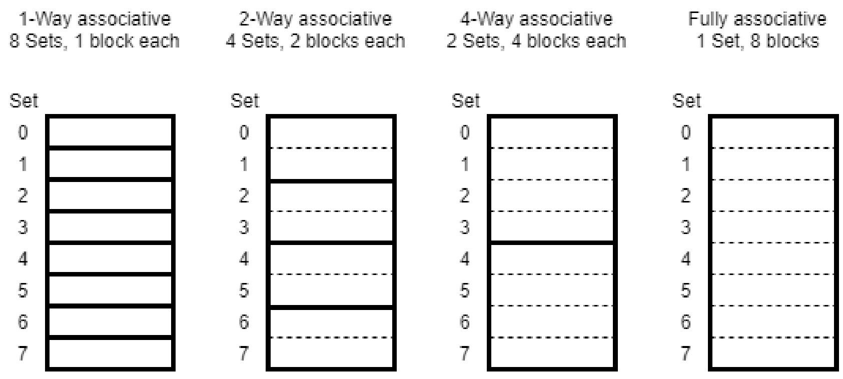


Figure 2.3.6: Associative cache organizations adapted from [10].

2.3.4 Replacement policies

When a cache is full and the controller needs to write new data to the cache, it has to determine which data to replace. For a direct mapped configuration this is simple. Each address is mapped to a single location in cache, so that location will be overwritten. For associative configurations each address can be mapped to multiple locations in cache so it is not obvious which data to overwrite. This is where replacement policies come in. There are different policies that, often on previous behavior, determine which location in cache will be overwritten. Some different policies will be discussed in this section.

Bélády's algorithm

The most efficient replacement policy would be the one that overwrites the data which is not needed for the longest amount of time in the future [11]. This is what the Bélády's algorithm proposes. However this would be a clairvoyant algorithm and in practice it is generally impossible to determine when data will be required again, making this not implementable in the real world. Since this would be the best policy, if not theoretical, it is mentioned. However for this research theoretical replacement policies like the Bélády's algorithm are excluded.

Random

One of the simplest policies is to randomize the cacheline to be replaced. This policy requires, unlike other policies, no information about the data currently stored in cache and is thus very simple to implement. It does however result in non-deterministic behavior and possibly varying performance results.

First In First Out (FIFO)

FIFO is a very well known algorithm in many different applications. It replaces the entry that is the longest in cache. To keep track of the different entry moments a simple queue can be implemented to determine which line to replace. This algorithm only requires information about the entry moment and disregards the usage when in cache. Alternatives on FIFO are, Last In First Out (LIFO) and First In Last Out (FILO). Which both have the opposite behavior of the FIFO replacement policy.

Least Frequently Used (LFU)

LFU not only keeps track of when data is written to cache, it also keeps track of when that data is accessed again. A counter keeps track of the number of accesses for each of the different

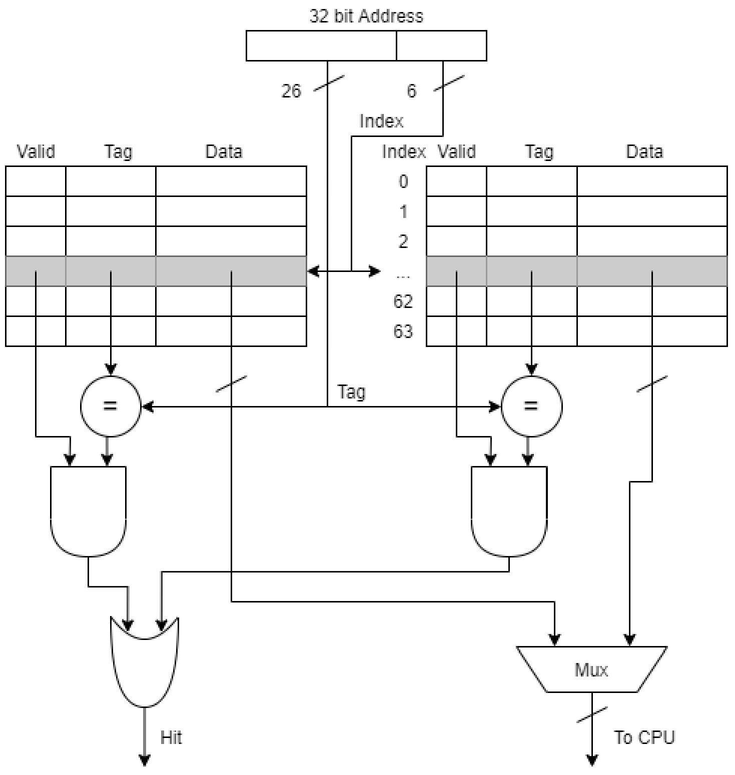


Figure 2.3.7: 2-Way associative cache memory logic adapted from [10]

cachelines in a set. When new data is written to the cache it is stored in the cacheline with the least number of usages at that moment.

Most Frequently Used (MFU)

MFU does the opposite of LFU. It also keeps track of the number of accesses for each cacheline but replaces the line that has been accessed most often. Intuitively this seems less efficient but like all replacement policies it is application dependent and there are cases where MFU is efficient.

Least Recently Used (LRU)

Where LFU and MFU keep track of how often a cacheline is accessed, LRU keeps track of when a cacheline is accessed. The least recently used cacheline in this case will be replaced. To keep track of when each line is last accessed is quite expensive. An age bit indicates for each line the order in which they are accessed. Every time a line is accessed the age bit of every line has to be updated making it time expensive.

Most Recently Used (MRU)

Just like LRU, MRU keeps track of when a cacheline was accessed. Instead of replacing the least recently used it replaces the most recently used cacheline. An MRU algorithm is generally most useful where older items are more likely to be accessed, which for instance happens in cyclic access patterns.

Time aware policies

On almost all mentioned policies are some variants besides the mentioned general implementation, that can be beneficial for specific use cases. These variations will not be discussed. However one group of replacement policies deserves a mention. Previous policies all determine replacements based on cache entry or accesses. There are however a lot of variations on these policies that are also time or performance aware and dynamically change during operation. An example is LFU with dynamic aging (LFUDA) where the age is of an absolute nature instead of aging relative to the other cachelines. Or an Adaptive Replacement Cache (ARC) which can dynamically change behavior between LFU and LRU based on performance. For this research the scope will be limited to the base replacement policies.

2.3.5 Performance metrics

For each application it can differ which aforementioned configuration and replacement policy performs best. In order to objectively compare different parameters, performance metrics are required. There are multiple metrics that can be used to compare different cache performance, including but not limited to the following metrics.

Hit Ratio

The most commonly performance metric of caches is the hit ratio, or hit/miss ratio. Every time a controller checks the cache to locate data there are two options. Either it is in cache, in which case it results in a cache hit, or it is not, resulting in a cache miss. When a cache returns a miss this means that the processor must retrieve the data from the main memory being slower. In order to exclude the total number of calls to the cache memory for this performance indicator, the ratio between hits and misses is taken. The higher the hit/miss ratio, the less amount of data transfers from the main memory are required, the better performing the cache.

Timing

The costs of a cache hit or miss will differ for different devices. So as the hit/miss ratio is a good general performance indicator for different configurations it does not take into account the device specific properties. For some devices the cost, in time, for a cache miss can be more severe, making extra overhead for more complex algorithms negligible. To incorporate these device specific properties one must look at the Average Access Time (AAT) it takes for a processor to access memory. This is also referred to as Average Memory Access Time (AMAT). This metric results in a real world performance indicator for a specific device.

The time it takes to access data is the time for a cache access and the time required to access data from main memory for a cache miss. With the access latency, the miss rate and the miss penalty known, the AAT can be calculated with the following formula:

$$AAT = T + MR \cdot MP$$

Where T is the latency to access the cache, MR is the miss rate of the cache and MP is the extra time required to access lower lying memory. In many CPUs a cache has multiple levels, L1 to L3. In that case the formula becomes:

$$AAT = T_{L1} + MR_{L1} \cdot MP_{L1}$$

Where MP_{L1} describes the extra time required to check the next level of the cache and can once again be described as:

$$MP_{L1} = T_{L2} + MR_{L2} \cdot MP_{L2}$$

This formula can recursively be expanded to all the further levels within the memory hierarchy of a design to get the final AAT or $AMAT$ [12].

Costs

Other comparison metrics are the costs. These costs can be defined in multiple ways; the amount of logic required, the complexity of the design or the amount of physical space. For this research these metrics are not the main comparison metrics and are therefore not further elaborated upon.

2.4 Tree-based algorithms

Examples of data-intensive algorithms that possibly suffer from a memory bottleneck are tree-based algorithms. Tree structures can become extensive and the computational complexity for tree-based algorithms often increase exponentially with the size of the tree, making hardware accelerating an effective approach. Considering the nature of some tree-based algorithms, the amount of data transfers possibly can be reduced by exploiting data relations captured in the tree structure to cache data. This is due to locality of data, not only temporal but also spatial. Tree-based algorithmic steps can be applied locally on subtrees without requiring the rest of the tree structure. In this case the entire subtree or part of it can be cached to anticipate next data needs exploiting the spatial locality of data in the tree structure. Similarly, the intermediate results of a leaf or subtree are often used in the next computation due to the child parent relation often present in tree structures. By caching intermediate data the temporal locality of data can be exploited. Examples of applications that can benefit from a locality utilizing accelerator structure can be found in the literature review in chapter 4.

Chapter 3

Research Method

In this chapter the objectives for this research are discussed as well as different possible research models that can be used with a discussion for the most suitable model.

3.1 Objectives

The purpose of this study is to reduce the effect of the memory bottleneck when using accelerator cards with dedicated memory for memory-bound tree-based algorithms. It will evaluate the performance of standard caching algorithms in combination with accelerator cards by comparing the number of data movements and computational times with and without caching. It will target tree-based algorithms that operate on subtrees by exploiting the spatial and temporal locality relations of data within the tree structures. The effectiveness of caching will be evaluated for the phylogenetic likelihood function. The tree structure employed in this application shows promising characteristics to benefit from caching. More details about the application and the selection can be found in the case selection section in section 4.3.

To summarize this research problem and the aims of this research the research question can be defined as:

How effective can caching be in alleviating the memory bottleneck when using an accelerator card for data-driven memory-bound tree-based algorithms?

To support the main research question some subquestions are defined.

- Can standard caching techniques be used for tree-based algorithms?
- What is the effect of caching on data movement and accelerator performance for the phylogenetic maximum likelihood function?
- What cache characteristics enable tree-based algorithms to benefit from caching when accelerator cards are deployed?

3.2 Research Model

The objectives give a clear goal for this research; Designing a software cache for a hardware accelerator. As the goal implies, this is a design problem making this research a design research opposed to a explanatory research. In a design research the goal is to create an artefact, a solution, with the aim of improving a problem context [13]. In order to make sure that this research is not only practically useful but still scientifically grounded, a research model for design research is used. Multiple models exists with the aim of ensuring this scientific solidity and practical relevance. Some of these models are discussed and the one best suited for this research is chosen.

In 2004 a theoretical framework aimed at design-science research within Information Systems (IS) was described by Hevner et al.[14]. This framework guides researchers on how to properly conduct, evaluate and communicate science research. It is a process of cycling through development and evaluation where every step must be grounded with relevant literature and also tested in the environment of final use. The Hevner framework is a strict framework and does not allow for different starting points but only from a problem-centered point.

A methodology that does support multiple starting points besides the problem-centered objective is the Design Science Research Method (DSRM) developed by Peffers [15]. The process model for DSRM can be seen in fig. 3.2.1 and clearly shows multiple research entry points instead of just the problem-centered initiation. It provides a less strict but still iterative process for conduction design research in a suggestive manner. Guiding research from the initial problem identification phase through design and development activities to an artefact evaluation. The need to provide the link between theory and the resulting artefact is not as strict but it is encouraged to ground design decisions with relevant literature knowledge. With this "relaxation" of rules the aim of DSRM is to make it easier for a researcher to develop artefacts while holding on to the well-accepted elements from other design research methods.

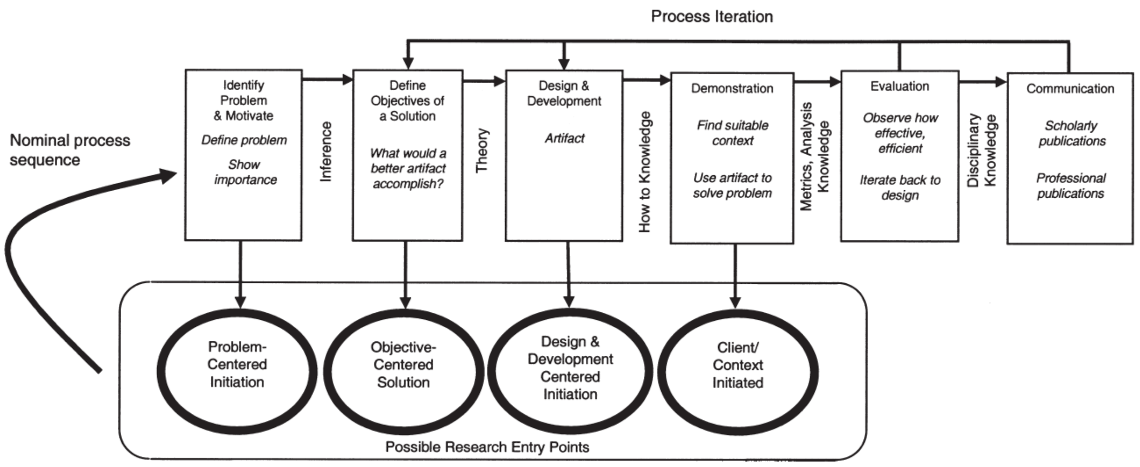

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 3.2.1: Process model as described by Peffer[15]

One major aspect that is missed in most design research methodologies according to Sein et al. is that they “fail to recognize that the [artefact] emerges from interaction with the organizational context even when its initial design is guided by the researchers’ intent” [16]. A lot of Information Systems are designed in collaboration with, or commissioned by, organizations and require the researchers to cooperate with industry partners. These industry partners often favor pragmatism above the scientific relevance and rigor. In there paper Sein et al. proposed an Action Design Research (ADR) methodology, taking aspects from the Action Research (AR) methodology and DR. This combinations allows for a more iteratively artefact building process, with organizational intervention and evaluation, making it a suitable methodology for industry driven research.

In order to pick the most suitable methodology for this research the main factor is the initiation point of this research. The initiation point is not problem-centered nor is it objective-centered but rather a design and development centered initiation. The memory bottleneck is already identified as the cause of a problem and the idea of creating a caching solution was the trigger of this research. Meaning the problem-, and objective-centered phases are already passed and this research is initiated in the design and development phase. Excluding Hevners methodology, which does not allow starting points other than the problem. Resulting in two remaining options, DSRM and ADR.

Even though the problem is identified in the industry this research itself is not conducted in cooperation or commissioned by an industry partner. This results in no need for extra intervention and evaluation. With that in mind DSRM is chosen as the research methodology.

The next step before starting the design phase is a literature review. The different aspects of the artefact implementation are explained and previous work is discussed. This is the starting point from where the artefact is designed and validated. Multiple validation cycles will ensure the functionality and the performance of the implementation.

Chapter 4

Literature review

This chapter outlines some of the relevant research done on the subject. First the use of hierarchical memory systems, as caching, are discussed. Secondly the potential for tree-based algorithms is shown. Lastly possible tree-based algorithms for this research are discussed.

4.1 Hardware accelerator

The potential platforms that could benefit from caching are included but not limited to the mentioned accelerator cards. The scope of this research will not include other hardware accelerator platforms.

4.2 Hierarchical memory applications

When thinking about the memory bottleneck, two obvious possible solutions spring to mind, increasing the transfer speed or decreasing the number of data transfers required. Assuming that applications already utilize the full bandwidth, it is very challenging to increase transfer rates without changing the hardware. Reducing the number of data transfers can be a solution but is highly dependent on the type of application. Efforts have been made in other scientific fields to reduce the number of data transfers for applications. For Deep Neural Networks (DNN) for instance multiple researches have come across a memory bottleneck in DNN research [17], [18], [19]. To reduce this effect a hierarchical memory management design is created. This memory management design is based on the different layers in DNN and shows promising results in reducing the number of data transfers [19]. So clearly a hierarchical memory can improve performance in data-intensive algorithms and is worth investigating.

4.3 Potential tree-based algorithms

In data-intensive tree-based algorithms the memory consumption and efficiency can be the limiting of hardware acceleration performance [2]. An example of this is the machine learning approach for phylogeny acceleration on FPGA [20]. Utilizing the availability of accelerator cards with a PCIe interface does improve the performance but is still limited by the data transfers speed between host and accelerator memories [21]. The hierarchical memory management designs, such as caching are not yet tested in a generic approach for data-intensive tree-based algorithms but the effectiveness on other applications show promising results.

Not all tree-based algorithms will benefit from caching. Based on multiple criteria tree-based algorithms are reviewed, the final results will validate if the set criteria were correctly chosen.

Since this research will try to reduce the memory bottleneck when using an accelerator card the applications first of all need to be computational intensive and have parallelization potential. These are properties to always look for when justifying the use of an accelerator card.

The other main criteria are derived from the aim to exploit the locality of data within tree structures, justifying the use of caching. For the temporal locality this results in finding algorithms with consecutive computations from child to parent node or vice versa. This allows for intermediate values to be cached. For the spatial aspect of locality tree-based algorithm that can perform computations on subtrees or branches locally without requiring information from the rest of the tree are reviewed. This way entire subtrees can be cached beforehand. With the criteria set, applications can be reviewed to be used in the rest of this research.

In the following sections three possible applications are elaborated upon as to how that algorithm works and why it shows potential for caching.

4.3.1 Phylogenetic maximum likelihood function

A phylogenetic tree is diagram to show evolutionary relations between different species based on genetic characteristics. It is an unbalanced binary tree structure where each leaf represents a different species. The purpose of the phylogenetic maximum likelihood function is finding the tree structure that fits the different genetic characteristics. However this becomes a computationally expensive task quickly when adding more species, for just 10 species there are more than 2 million different topologies [22]. In short, the maximum likelihood function determines the likelihood of any given tree with a summation of the probabilities of the changes in nucleotides from node to node.

In 1981 Joseph Felsenstein published a pruning and regrafting approach on the phylogenetic maximum likelihood function [22]. In this research he stated that the probability on a given tree with a given set of data can be computed node by node. At the end of the computation the product of the probabilities can be taken across sites to find the likelihood of that tree. This way transformations in a tree can be made and locally computed. The transformations proposed were subtree pruning and regrafting. Relocating entire subtrees in the complete tree structure.

With this node to node attribute, the algorithm can compute the likelihood of a subtree without requiring the rest of the tree structure. This together with the recursiveness of the algorithm creating a relation within tree between child and parent nodes, creates a locality between the data that potentially could be exploited with caching.

4.3.2 Polymer simulation

A polymer is a generalized term for a string consisting out of many monomers that contain many properties based on the structure and form. A good example are protein. The structure of the protein contains information about the energy and computations with protein, protein folding, are part of medical research. Protein folding is a challenging objective in biochemically research. Although physical principles are known, the complexity of protein makes an accurate analysis by modeling the folding process extremely difficult [23].

Recent study shows that polymer strings can be described with a balanced binary tree structure where the leaves represent the monomers and the connecting nodes represent the physical transformation between two child nodes or leaves. With this approach energy levels in polymers can be described on monomer level and via the nodes can be summed to a total energy level for that polymer at the root [24].

This results in an algorithm that computes the total energy based on the design of a tree similar to the phylogenetic maximum likelihood function. The relational data between parent and child nodes creates an opportunity for caching.

4.3.3 Gradient boosted decision trees

Decision trees are often used in decision analysis, identifying a most likely strategy to reach a certain goal, it is however also a popular as a machine learning tool. Gradient Boosted Decision Tree (GBDT) is an ensemble algorithm of decision trees in which the trees are trained in sequence. For each iteration, a new decision tree is learned by fitting the residual (also known as negative gradient). In other words, GBDT trains decision trees in an incremental way: at tree n fitting the residual value of tree $n-1$ [25].

After training, during inference, the data is passed through all decision trees, each is multiplied by a set factor and finally combined to a single decision. The main cost of GBDT models lies in the tree learning stage [26].

GBDT models have been accelerator with hardware accelerator cards. Here the overhead communication of transferring the data to and from the accelerator proved to be high and reducing the performance of the accelerator [26].

The dependency of a tree on the preceding trees residual value during the training process creates relational data that could be cached between each tree.

4.4 Literature review conclusion

From the literature review some conclusions can be drawn. Firstly that a hierarchical memory management systems has proved to increase performance in memory-bottlenecked systems. Secondly the fact that there are tree-based algorithms that show this bottleneck but also the potential for exploiting locality of data. This makes the implementation of cache a potential improvement for tree-based algorithms. In the design a higher memory level is created in the form of a software cache. Its performance will be tested against the phylogenetic maximum likelihood algorithm.

Chapter 5

Research Design

In this chapter the design of the software cache is described. First the different general design principles are discussed in section 5.2. After that the development platform and architecture are described in section 5.3. Next the actual implementation, its challenges and solutions and code design are described in section 5.4. Lastly the interface is elaborated upon in section 5.5.

5.1 Cache architecture

In section 2.3 the standard architectures for caches are described. For this implementation the architecture slightly differs. The cache controller and cache memory are separated. The host processor is the master for all communications in this system, it initiates all data transfers both to and from the accelerator card. For that reason the cache control will run on the host processor. The actual data however needs to be stored on the accelerator memory and is thus separated from the rest of the cache controller. This separation in cache can be seen in fig. 5.1.1. The host processor keeps track of the different entries in the cache memory and now has an additional field containing a pointer pointing to the location of the data in the cache within the accelerator card's memory space. In fig. 5.1.2 a block diagram of the new system is shown. It is similar to the original block diagram shown in fig. 1.1.1 however the memory space on the accelerator card holds the cache memory and the host processor is running the cache control process.

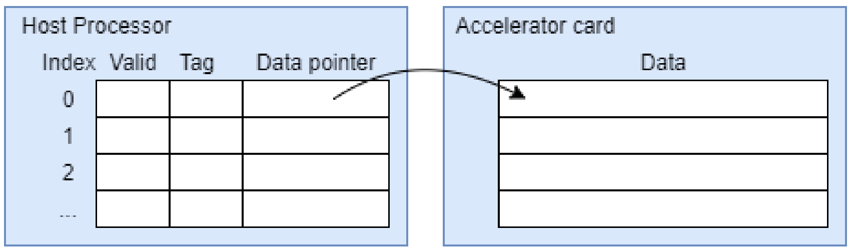


Figure 5.1.1: Cache separation

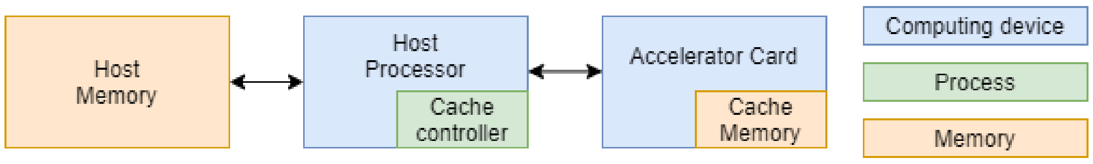


Figure 5.1.2: Block diagram of the architecture for host and accelerator with caching.

5.2 Generic Application

The goal of this research is to design a software cache for accelerator cards to increase performance for tree-based algorithms by reducing data movement. This is a generic goal. There are multiple devices that class under the name accelerator card. The same is true for tree-based algorithms. In order to design a software cache to accommodate the different platforms and algorithms it has to be generic. The design choices made to achieve a generic implementation for different platforms is explained in section 5.3, where the development platform and architecture are discussed.

To make sure the design supports different algorithms there must be some level of control for the user. Like mentioned in section 2.3 there are multiple different configurations and replacement policies that define the functionality of a cache. For each application it will differ which combination of cache configuration and replacement policy will perform best. So the design must include multiple configurations and replacement policies and allow the developer to choose. The design must also allow the user to define a block size for the data to be stored. For tree-based algorithms the most likely block size will be the data size for an individual node. Since this caching method is aimed at data-intensive algorithms this most likely will result in large block sizes in the order of megabytes or even gigabytes. The block size is set by the user and does not have to be the size of a node.

Depending on the different platforms used the amount of on board memory will differ. So the design must accommodate different sizes. The way that is done is by defining the number of cachelines the cache will have. For indexing purposes discussed in section 2.3.1 the number of cachelines is limited to powers of two.

By doing so the host addresses can directly be mapped to certain cachelines without adding a complex controller that increases the cache control time. This does have some downsides. In a worst case scenario this allows up to 50% of memory space on the accelerator card being unusable. When a accelerator for example has enough space for 127 cachelines. This would mean that the maximum number of bits that map to a cacheline would be 6, resulting in 64 cachelines. When 7 bits are used there are 128 indices and one of them is not pointing to a cacheline. In this example only 64 of 127 possible cachelines can be used. For larger cachesize this unused memory space can grow up to 50%. Since in the expected use cases the number of cachelines will be small the simpler, faster, controller is chosen over the more complex controller, resulting in the limitation in number of cachelines.

The last setting a user can control is the tag size. In section 2.3.1 the tag is defined as the address in main memory minus the number of bits used for mapping to the cache index. In almost all cases this will suffice. If the data for an algorithm is stored in a small portion of the main memory less bits can suffice in uniquely identifying each data entry. In that case the user can pick a smaller tag size in order to save memory space. In practice the memory space required for the tag is an order of magnitude smaller than the block size and won't make much difference overall but it ensures that only the necessary amount of memory is allocated. In table 5.2.1 is an overview list of all settings that can be set by the user.

5.3 Development platform and architecture

During the design process not all different accelerator platforms discussed in section 2.1 are used for verification. A platform and the design tools are selected and discussed below.

Table 5.2.1: Different settings for the cache available to the user.

Setting	Value
Cache configuration	{Direct mapped, 2-Way associative, 4-Way associative, Fully associative}
Replacement policy	{Random, FIFO, LRU, MRU, LFU, MFU}
Block size	Number of bytes
Cache size	Number of cachelines
Tag size	Number of bits

5.3.1 OpenCL

The cache must support multiple accelerator cards. There already exist many implementations of cross platform applications. With Open Computing Language (OpenCL) development times are significantly reduced compared to traditional low level languages resulting in wide spread use of OpenCL [27]. OpenCL is a framework for writing code that is executed on multiple platforms. It is supported by CPUs, GPUs, FPGAs and other hardware accelerators, making it an ideal tool for this application. The code for the cache will include OpenCL to control the cross platform communication. This way it can be implemented on any OpenCL supported device.

Not only will the code include OpenCL code, the goal is to also create an OpenCL like interface for the caching software. This way it is understandable and implementable for other users without having the understanding of the entire code.

5.3.2 Test platform

By including the OpenCL it is not necessary to develop for each platform individually and a single platform can be used for development. In this case the design will first be implemented for an FPGA based accelerator card. More specifically the Alveo card family by Xilinx [28]. The Alveo cards offer Gen3x16 PCIe interfacing and on board memory space. Besides the hardware Xilinx also offers development tools such as Vitis to use Alveo cards with a High Level Synthesis (HLS) approach [29] to create cross platform software. With Vitis there is no need to code the behavior for the FPGA in a Hardware Description Language (HDL) but code can be developed on a higher abstraction level. This way the code is much more readable for others and easier to make changes during development. Vitis supports both regular C language as well as C++. For the design C is chosen as it is the more basic language and the extra functionality added by C++ is not directly needed. The Vitis tool will create HDL code based on the C code provided in both VHDL and Verilog form.

A final added benefit of targeting the Alveo cards by Xilinx is that they are available to be used through the Amazon web services (AWS). AWS offers the possibility to rent computing power on servers from Amazon. AWS uses a derivative of the Alveo platform, that is also supported by the Xilinx tools, making it possible to test the implementation on the AWS server without having to invest in a expensive card.

5.3.3 Development

During the first stage of development the accelerator card is not included. Since the code is written in C the behavior can be tested on a regular CPU without having to convert it to HDL. This is how the code is tested during development. A separate memory space on the main memory is allocated to mimic the external memory on the accelerator card. This way all the different functionalities of the caching software can be verified on a single system.

Since the code is written in plain C code, the verification of the code can also be written in C, making it simple to do. During development testbenches are written in C for each function within the code. With the help of makefiles and shell scripts these tests can easily be deployed making it a very user-friendly method of developing.

After all functionality of the cache has been verified and evaluated in C, the transition to a cross platform implementation is made. At that stage the OpenCL support is added and the Vitis tools are introduced to create HDL for the accelerator card.

5.4 Implementation

During the implementation of the design some more choices had to be made. The different approaches and final implementation strategies are elaborated upon in this section.

5.4.1 Memory allocation

For the cache controller memory needs to be allocated on both the host processor and accelerator card. This memory space accommodates the data but also stores the cache controller information and other required information. Different memory allocation types were considered during the design process which are discussed here.

Offsets

The initial design for the cache was a monolithic memory allocation encapsulating the entirety of the cache. This approach is similar to how a hardware cache is designed. The idea was to use offsets within this memory space to identify different cachelines or fields within this memory space. In fig. 5.4.1 the structure of this approach can be seen.

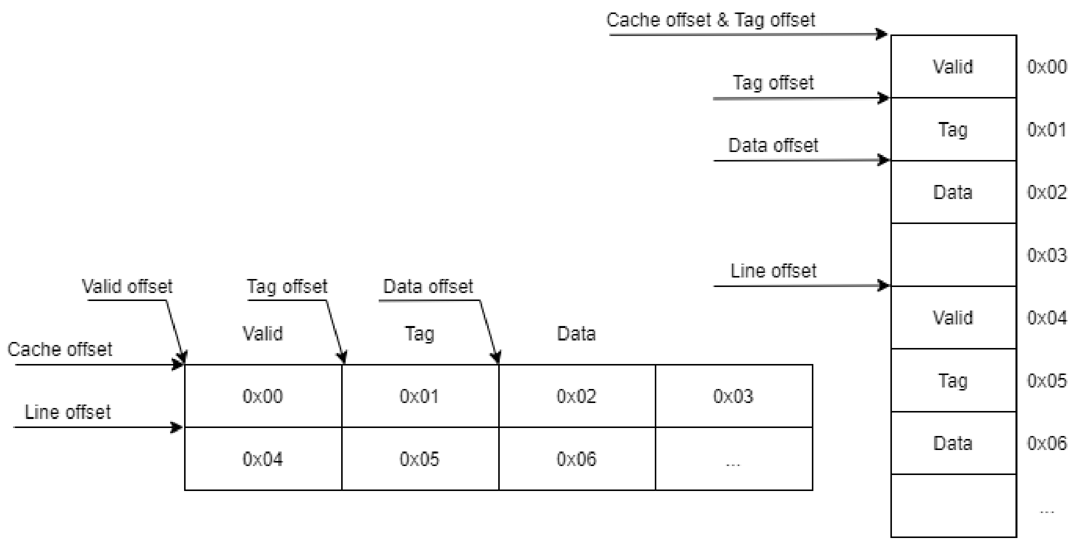


Figure 5.4.1: Offsets used in a monolithic memory allocation

Here the `Cache offset` points to the start of the memory block in the host memory, which in this case is `0x00`. Since the cacheline starts with the valid bit no `Valid offset` is required but for completeness it is included in the figure. With these constant offsets the different parts of the cache can be identified with some simple formulas. The formulas for the tag and the data for instance can be seen in eq. (5.4.1) and eq. (5.4.2) respectively.

$$\text{tag} = \text{Cache offset} + (\text{index} * \text{Line offset}) + \text{Tag offset} \tag{5.4.1}$$

$$\text{data} = \text{Cache offset} + (\text{index} * \text{Line offset}) + \text{Data offset} \tag{5.4.2}$$

By having just a single large memory allocation it is easy to free up again preventing possible memory leaks. With only the pointer to the cache the entire memory space can be freed. There are however downsides to this approach. The first of these downsides being the readability and adaptability. Adjustments to the code can be made by altering the offsets but this is not an intuitive approach. It does not support having multiple different caches since the offsets are fixed. All different caches will have the same design and functionality. The largest downside however is the lack of flexibility for the allocation. A monolithic memory allocation requires a large block of memory available. This is not efficient since it leaves no room for the processor to dynamically allocate different, smaller, memory spaces to optimize memory usage.

An even bigger concern for the final design part of the memory, the data, does not have to be allocated on the host memory but only on the accelerator card. This would mean that the offsets have to be across different memory spaces making it very tricky and prone to mistakes if possible at all. So the monolithic memory allocation with offsets is not suitable for this design.

Structs

Another approach is the use of custom structs. This allows for much more flexibility during the design and readability for any user. With these structs an object oriented approach can be mimicked and parts of the cache can easily be accessed and adapted. The main benefit is the separate memory allocations for different parts of the cache. This allows the processor to dynamically allocate smaller chunks of memory and also allows easier separation across different platforms. To achieve this flexibility some custom types and structs are defined which can be seen below.

Two enumeration types are defined to support the different configurations and replacement policies. These enumerations are not only for readability but also make sure that the user provides a configuration and replacement policy that is supported by the code.

```
typedef enum CacheConfiguration_t {
    direct_mapped,
    two_way,
    four_way,
    fully_associative
} config
```

```
typedef enum ReplacementPolicy_t {
    random_RP,
    fifo_RP,
    lru_RP,
    mru_RP,
    lfu_RP,
    mfu_RP
} policy
```

With these types defined, the structs can be defined as well. First a struct for an individual cacheline. This struct contains the valid bit, tag, data and some metadata if required. An example of this metadata is the number of usages of that specific cacheline for the replacement policy. Entries in the struct can simply be accessed as follows
`valid=CacheLine_t.valid`

The `CacheLine_t` struct can be seen below. The `deviceData` is of type `cl_mem` which is a pointer to the data stored on the accelerator defined within OpenCL.

```
typedef struct CacheLine_t {
    bool valid;
    void* tag;
    cl_mem deviceData;
    struct MetaData_t* metaData;
} CacheLine_t
```

The main struct for the cache can now be defined. First this struct contains all the parameters of the cache given by the user. The second variable in this struct is a double pointer of type `CacheLine_t`. This double pointer is used to create a two-dimensional array of cachelines. This way a distinction between sets can be made. By dereferencing the first pointer the location of a set in memory can be identified, by then dereferencing the second pointer the location of a specific cacheline within that set can be found. An example on how to access variables in a cacheline with this type: `valid=Cache_t.cachelines[Set][Way].valid`

For a direct mapped or fully associative implementation the array is limited to just one dimension and either the `Set` is fixed to zero or the `Way` is fixed to zero. Part of the `Cache_t` struct can be seen below. The actual implemented struct contains more variables than are depicted here but for these are the basic variables required for the working principle of the cache. It is very simple to just add variables to one of the structs making the development process easier than it would have been with the monolithic memory allocation.

```
typedef struct Cache_t {
    int tagSize;
    int dataSize;
    int numberOfLinesPerSet;
    int numberOfSets;
    enum CacheConfiguration_t config;
    CacheLine_t** cacheLine;
    enum ReplacementPolicy_t policy;
} Cache_t
```

The use of structs does have some downsides. Since every variable in the structs has a separate memory allocation a custom free function has to be made. In section 5.5 more details about this custom function are given. The positives outweigh the negatives for structs so the final design uses these custom structs.

5.4.2 Indexing

In section 2.3.1 the indexing method is explained. For a cache with 2^n cachelines the last n bits of the memory address on the host memory is used to index the cache memory. To achieve this functionality a bitmask is created based on the number of cachelines. First the number of bits for the index, the `indexSize` needs to be determined. For a direct mapped cache this is done by taking the $\log_2(numberOfCacheLines)$. Base two logarithm is not supported directly in C so to determine the `indexSize` eq. (5.4.3) is used.

$$indexSize = \frac{\log_{10}(numberOfCacheLines)}{\log_{10}(2)} \tag{5.4.3}$$

Now the number of bits is known a mask is created to use in the code. With the eq. (5.4.4) the bitmask is determined in decimal form. The binary format of that number will be a string of ones the length of `indexSize`. This bitmask is applied with a bitwise `AND` operator to extract the index from the main memory address. The bitmask is stored as integer in the `Cache_t` struct. This way the processor does not have to compute the same value every time the cache is called.

$$indexBitMask = 2^{indexSize} - 1 \tag{5.4.4}$$

In section 2.3.1 a possible problem with this direct mapping method is already explained. When the block size in main memory and in cache memory are not the same size part of the cache memory space will never be used. This is actually very likely of happening. An example; When in main memory the data for each node requires 4 bytes, that data will be stored in a single cacheline with 4 bytes of memory for the data. In fig. 5.4.2 this effect can be clearly seen, just one of the cachelines in this case is usable.

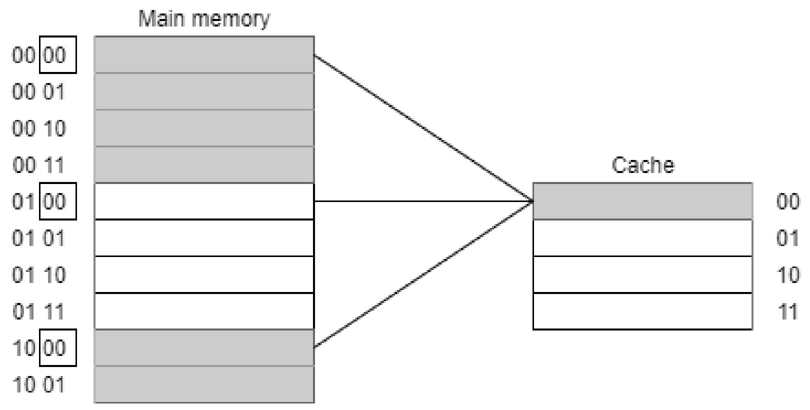


Figure 5.4.2: Direct cache indexing for a data size of 4 bytes.

To prevent this from happening a clever trick is introduced. The reason that this happens is that for an alternating data pattern the last bits of the memory address will not change. To regain full mapping potential the bitmask will be shifted to the part of the address that does change. Since the cache only supports a constant data size this trick can be applied. Going back to the example of 4 bytes of data per node, the bitmask will be shifted two bits to the left and the entirety of the cache is usable again as can be seen in fig. 5.4.3.

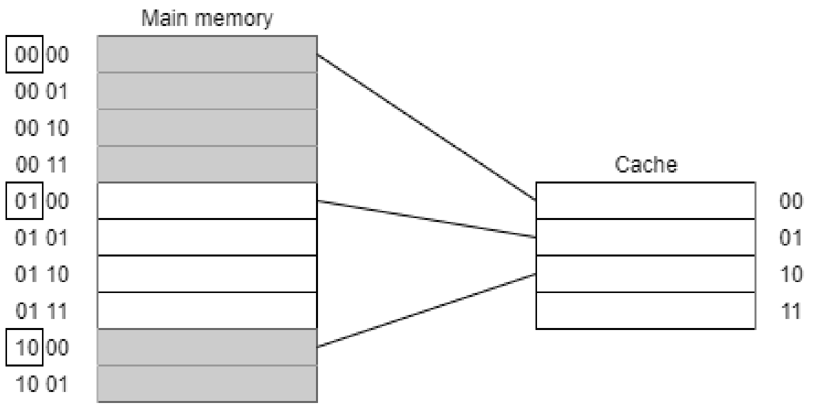


Figure 5.4.3: Direct cache indexing for a data size of 4 bytes with a shifted bitmask.

To find the number of bits that do not change at the LSB side of the memory address the loop in listing 5.1 is implemented:

Listing 5.1: Loop to find the number of bits to shift

```
for (addressBitShift = 0; addressBitShift < dataSize;
    addressBitShift++) {
    if ((dataSize % (int)pow(2, addressBitShift + 1)) != 0)
        break;
}
```

The resulting value for `addressBitShift` is the amount of bits that the bitmask needs to be shifted over the memory address and is just like the bitmask stored as integer value in the `Cache_t` struct. The complete process of extracting the index from a memory address is put in a function that returns an index in integer form. That function is shown below.

```
int GetIndex(int indexBitMask, int addressBitShift, void*
    hostAddress) {
    return (int)(((intptr_t)hostAddress >> addressBitShift) &
        indexBitMask);
}
```

5.5 Interface

As mentioned in section 5.3.1 the idea is to create a set of functions that adhere to the OpenCL naming convention to make implementation for users as simple as possible. This cannot be achieved for all functions since some have no counterpart in OpenCL such as the `CreateCache` and `FreeCache` functions. Since the code for the software cache is written in `C` the OpenCL `C` syntax is used. Below are the different interface functions listed and explained. The code does contain more functions than listed but those functions are not part of the API and for internal use only.

The use of the interface in a real application can be seen in section 7.5. Here code with cache is compared to a regular OpenCL use.

CreateCache

The first function is an initialization function. This creates a cache controller. The code is written in C so there are no objects but the behavior of this function would be the same as creating an instance of a class in object oriented programming languages. This function allocates memory space based on the arguments given and returns a pointer of type `Cache_t` as defined in the header file, pointing to that specific memory space. The allocation happens on both the accelerator card, for the actual data, as well as on the host memory for all the other data. The pointer returned by this function must be used as argument for the other functions for the controller to locate the allocated memory space. The benefit of this approach is that multiple instances of caches can exist and function at the same time. Each being uniquely identified by a pointer of type `Cache_t`. Simply provide the correct pointer as an argument in the function call.

For the initialization the function parameters are required. In section 5.2 the different parameters that are supported by this cache to maintain the generic design are listed. All those parameters are required as arguments at initialization. At initialization the code prints the information back to the terminal so the user can see the total amount of allocated memory space, including the overhead memory space, as well as the amount of memory allocated for the actual data.

Just like an object oriented programming language with an initialization of a object also the destruction at the end of use is important. The `CreateCache` function should always be used in combination with the `FreeCache` function that is mentioned below.

```
        struct Cache_t* CreateCache(
            int numberOfCacheLines,
            int dataSize,
            int tagSize,
            enum CacheConfiguration_t config,
            enum ReplacementPolicy_t policy)
```

FreeCache

As mentioned in the section above the `FreeCache` function must always be used when creating a cache. In the initialization function memory space is allocated for the different fields of the cache. That allocated memory space must be freed up again in to prevent a memory leak. The regular `free` function defined in C is not enough in this case. Since the memory allocation in the `CreateCache` function is a recursive allocation process, that allocates separate memory blocks based on the provided arguments there is no direct way to keep track of each allocation. To do this the `FreeCache` function is provided. Simply provide the `Cache_t` pointer as an argument and every individual allocated memory block will be freed.

```
        void FreeCache(
            struct Cache_t* cachePtr)
```

clCreateCacheBuffer

When using OpenCL the `clCreateBuffer` function is used to point data from the host memory to a `cl_mem` object on the accelerator memory. In order to use the software cache the `clCreateCacheBuffer` can be used in a very similar fashion. This function first checks if the data is already in the cache memory and if so returns the `cl_mem` object pointing to that location. If the data is not in cache it will find a space and transfer the data over before returning a `cl_mem` object pointing to the new location of the data. The returned `cl_mem` object can be set as a kernel argument before starting the kernel.

This function sets all the attributes of the cacheline, such as the tag and valid bit but also updates the accessed order for the replacement policies. Below the `clCreateCacheBuffer` function can be seen. The only addition in syntax with respect to the regular OpenCL `clCreateBuffer` function is the addition of the pointer to the cache.

```
        cl_mem clCreateCacheBuffer (
            cl_context context,
            cl_mem_flags flags,
            size_t size,
            void *host_ptr,
            cl_int *errcode_ret,
            struct Cache_t* cachePtr)
```

Depending on the flags given as argument the function can slightly change. By default, the flags `CL_MEM_READ_WRITE` and `CL_MEM_COPY_HOST_PTR` are given as arguments for this function. This tells the cache that the data is only to be transferred when not in cache already.

There are however instances when an entry is already in cache but the data is outdated and needs to be overwritten and thus transferred to cache. To do this the `CL_MEM_COPY_HOST_PTR` flag must be left out. This way data is always written to the cache. This is also required for the output data.

clEnqueueReadCacheBuffer

The last functions is used to retrieve data from the cache back to the host memory. This function is an adaption on the `clEnqueueReadBuffer` from OpenCL. The difference however is that no `cl_mem` object is given as argument since the cache determines where the data to be read back is stored. When a cache entry is requested while not in cache the function will return `1` and no data is retrieved from the accelerator.

The main benefit of this function opposed to its OpenCL counterpart is the option to retrieve any data in the cache memory. With `clEnqueueReadBuffer` only the most recent calculated data can be retrieved from the accelerator memory. With this function any data in cache can be retrieved by simply providing the pointer to the data on the host memory.

```
int clEnqueueReadCacheBuffer (
    cl_command_queue command_queue,
    cl_bool blocking_read ,
    size_t offset ,
    size_t size ,
    void *host_ptr ,
    cl_uint num_events_in_wait_list ,
    const cl_event *event_wait_list ,
    cl_event *event ,
    struct Cache_t* cachePtr)
```

5.5.1 Supported configurations and replacement policies

In section 2.3 different architectures for caches are described. Based on the configuration and the replacement policy the functionality of the cache differs. To keep this research feasible not every possible architecture is implemented.

Configurations

Three different types are described in the background, direct mapped, set associative and fully associative caches. All three different configurations are implemented in the code. The Set associative configuration however is limited to 2-Way set associative and 4-Way set associative. Others can be added in future versions with relative ease. But for the goal of this research the 2,- and 4-Way set associative caches provided enough functionality for comparison with the direct mapped and fully associative cache for the reason that in the practical implementation cache size most likely will be very small, no larger than 8 or 16 lines. An overview list of the different supported configurations is seen below.

- Direct mapped
- 2-Way set associative
- 4-Way set associative
- Fully associative

Replacement policies

As mentioned in section 2.3.4 the policies implemented in this research is limited to the base policies, so no variations or combinations of policies. The goal of this research is to evaluate if the memory bottleneck can be reduced with a software cache, resulting more in a proof of concept design instead of a highly optimized implementation. The current implemented replacement policies are:

- Random
- FIFO
- LFU
- MFU
- LRU
- MRU

For future research other, more complex, policies can be added to possibly improve performance further.

Chapter 6

Evaluation method

In this chapter the evaluation method for the software cache is described. For the evaluation the general cache performance metrics described in section 2.3.5 will be used as well as application specific metrics. The design is intended to be generic but the tests will be performed on one real-world scientific application as a proof of concept. First the different tools that are used for the testing are described in section 6.1.2. Next the different tests and their respective goals are described. And finally some notes on the reliability of these tests are given.

6.1 Use case

As mentioned before the effectiveness of a cache and the different configurations is highly dependent on the application. To evaluate the performance of the software cache more realistically a real world tree-based algorithm application is used. In the section 4.4 the Phylogenetic Likelihood Function (PLF) [22] is chosen for this evaluation phase.

6.1.1 Phylogenetic likelihood function

This algorithm uses matrix multiplications in order to determine the likelihood of a part of a tree structure. It uses double precision values for each of the alignment patterns within a given DNA sequence. Each DNA sequence can easily contain more than 100.000 alignment patterns with 16 double precision values per pattern making it data intensive.

6.1.2 Input

External tools are used to create input data for the testing of PLF. The different tools and how these are used are described below.

Sequence generation

Two tools are used to create simulated sequence data, `ms` and `seq-gen`. `ms` is a tool that can generate a sample of a neutral model [30]. Next the `seq-gen` tool can create multiple DNA sequences based on that generated sample and some given parameters [31].

For this research the exact type of phylogenetic data is not of concern so simple data is generated with the following commands.

```
ms X 1 -T | tail +4 | grep -v // >treefile_name
```

Where `X` is the number of different sequences, in other words nodes in the tree, `-T` specifies the output type, `tail +4` and `grep -v` remove comments and other information from the output file and finally `>treefile_name` represents the output file. `X` in this case is variable and will change for different tests. Next is the sequence creation with the `seq-gen` tool.

```
      seq-gen -mHKY -l 1000 -s .2 <treefile_name >seqfile_name.phy
```

Here `-mHKY` specifies the used model, `-l 1000` is the number of alignment sites for the sequences, `-s .2` sets the variation θ to .2 per base pair and then the names of the in- and output files are set.

6.1.3 RAxML

There is a tool created called RAxML [32], which stands for Randomized Axelerated Maximum Likelihood. This tool, determines the likelihood of different phylogenetic trees for a given phylogenetic data sets, just like created in the previous section. This tool will be used as a base for testing the cache software.

To run RAxML is quite computationally intensive so to speed up testing the tool is only used once and during that execution the memory access trace is extracted. With this trace many tests can be executed without being limited by long execution times. The code has a continuous loop in which it provides three memory addresses. The first two addresses point to the data of the child nodes. The third address point to the data from the parent node, here the result of the calculation will be stored.

The access pattern is stored in a tracefile. A small example part of a trace can be seen in listing 6.1. Here the first two addresses of each line are pointers to the child nodes and the last address is a pointer where the result must be stored. The `width` given at the top represents the length of the genomic sequences to be processed.

Listing 6.1: The beginning of a memory trace

```
width 156
0x2039d94      0x2039e30      0x21f4940      tt
0x203a0a0      0x203a13c      0x21f9750      tt
0x203a004      0x21f9750      0x21fe560      ti
0x2039f68      0x21fe560      0x2203370      ii
...
```

As can be seen each line also contains two letters, which can be a `t` or an `i`. This letter indicates if the respective childnode is a tip node or an inner node in the tree. This information will be used later on.

The RAxML tool is called with the following parameters:

```
      raxmlHPC -m GTRGAMMA -s seqfile_name.phy -n T1
```

Here `-m GTRGAMMA` sets the computation model used, `-s seqfile_name.phy` defines the input file and `-n T1` defines the output file. the output files however are of no concern since the only interesting result at this stage is the tracefile that is generated.

6.2 Experimental setup

In this section the different tests that are used to compare the influences of the cache characteristics are described. First some generic design choices that apply to all tests are described in section 6.2.1. Next the different tests and their goal are described. Finally, in section 6.2.4 some notes are given on the reliability of the tests and how they relate to a real world application.

6.2.1 General test design and constraints

Multiple different tests for the PLF will be executed. All will have some similarities based on the way PLF works. Those similarities result in test design choices stretching across all tests that are explained here.

Node differentiation

In a phylogenetic tree the leaf nodes, from here on called tip nodes, represent different DNA sequence from different species. The inner nodes however represent something else, namely a series of probability vectors. This differentiation between tip and inner nodes is actually important since there is a different amount of data stored for each type of node. All cachelines within a cache have a fixed size data field. It is possible to store smaller amounts of data in a larger cacheline but that is not efficient. In the case of PLF the inner nodes contain 128 times the number of bytes as the tip nodes, making it inefficient to store the data of both nodes in a similar sized data field.

In this case it would make more sense to create two different cache instances with different sizes, one for the tip nodes and one for the inner nodes. That is why, as mentioned in section 6.1.3, the trace includes the type of node that is called so the controller knows which cache to use. This separation in two different caches is a great way to more efficiently use the memory available on the accelerator card but it also has some downsides. The main downside is the doubling of the state space. To cope with this larger state space the performance of the tests are evaluated per cache since the caches do not influence each other.

Traces

The tracefiles are created with the tools mentioned in section 6.1.2. These tracefiles are based on the generated phylogenetic data. The way this data is generated influences the tracefile. There are two main settings that are important for this research, the number of DNA sequences and the number of alignment sites per sequence. First, the number of sequences directly translates to the number of tip nodes in the tree. Each tip node represents a DNA sequence. By increasing the number of sequences the number of different data blocks that need to be transferred to the accelerator increases. To test the performance of the cache the number of DNA sequences is varied between the following values:

- 100 sequences
- 250 sequences
- 500 sequences
- 1000 sequences

These values are chosen to still create a realistic behavior without creating unfeasible long computation times. The mentioned number of alignment sites determines the data size per node required for the application. A higher number of alignment sites creates larger data sizes but also much longer computation times for RAxML. For testing this number is kept as low as possible but consistent for all DNA sequences. The way realistic data is generated means that the number of alignment sites must be larger than the number of DNA sequences to guarantee uniqueness of each sequence. This is required to get a realistic memory access pattern so the number of alignment sites is set to 1000. In real phylogenetic data this would be much larger in the order of 10.000 alignment sites or more but for the access pattern it is less important.

Lastly all traces extracted from RAxML are limited to 1.000.000 memory calls to have a fair comparison for different number of nodes during testing.

Cache size

The number of inner nodes depends on the number of tip nodes. Since a phylogenetic tree is a binary tree with an unknown root, it is represented as an unrooted tree. This means that for n tip nodes there are $n - 2$ inner nodes. For RAxML this is not entirely the case. For computation RAxML introduces an imaginary root in the tree. This means that an extra inner node is created and the number of inner nodes is $n - 1$.

Since both numbers are more or less equal both cache sizes will be set to the same number of cachelines. Even though the data size for inner nodes is 128 times greater this simplification is made as first step to reduce the state space.

Other constraints

Finally some other constraints are made. First all testing will still be performed entirely in C code on a regular CPU. For the initial state space reduction the cache is not yet operated across platforms to reduce complexity.

Since the actual data during testing is not of importance the data transfers are excluded from the tests. In this case only the controller is updated but no actual data is transferred to reduce execution times.

For each computation the accelerator requires two inputs to produce an output. For the inputs first the cache is checked before a transfer is made. The output however cannot be in cache already since the current value is not known beforehand. After the computation the output will always be stored in cache, more specifically the cache for inner nodes. From there it will be written through to the host memory.

6.2.2 Performance metrics

In section 3.1 the different objectives of this research are mentioned. One of them being: *What cache characteristics enable tree-based algorithms to benefit from caching when accelerator cards are deployed?* In order to find an answer to this question tests are performed, each time varying the settings of the cache and thus the characteristics.

There are three main characteristics that can be configured in the designed caching software:

- Cache size
- Level of associativity
- Replacement policy

In most implementations the cache size will be dictated by the available memory space on the accelerator card. It also depends more on the ratio between number of cachelines and number of data entries on the host than it depends on the absolute number of cachelines. Because of that the focus in the tests will lay on the level of associativity and the replacement policy. However the cache size is not excluded from the tests, every test will be performed with different cache sizes so the influence of the ratio can be determined.

To evaluate the different levels of associativity and replacement policies performance metrics are used. In section 2.3.5 different generic cache performance metrics are described. There are however also metrics more specific to a use case where application or platform specific parameters are used. The sections 6.2.2 to 6.2.3 describe the different performance metrics used for the comparison.

Once the influence of the different cache characteristics is known the same results can be used to answer the other research questions.

Hit ratio

The first test covers the hit/miss ratio. Here the testbench follows the memory access pattern from the tracefile. For each line in the trace the controller checks in the correct cache if the two childnodes for that computation are available otherwise it stores it in cache. The output is always stored in cache. Both cache controllers keep track of the number of simulated data transfers and the testbench keeps track of number of memory access for each cache. The gathered numbers are used to find the ratio with the following formula.

$$\text{Hit Ratio} = (1 - \frac{\text{Number of data transfers}}{\text{Number of memory calls}}) * 100\%$$

Since the output always has to be written to the cache this is not considered as a cache miss and thus not included in the hit/miss ratio. This test will be executed for all possible combinations of configurations and policies, all different sequence lengths and for different cache sizes varying from 4 cachelines to 16.

Overhead timing

Next performance metric is the overhead timing costs introduced by the cache. This overhead includes the checking of the cache and updating accessed orders for replacement policies but excludes the data transfers. Associative caches have to loop through the entire set to find an entry opposed to the direct mapped which has no loop. Especially for larger cache sizes this extra overhead timing for associative caches could make a difference.

To compare different overhead timing a similar test as for the hit/miss ratio is performed. Memory accesses are simulated with the tracefiles. Since the timing capability within C is discrete with quite large steps it is not possible to time each cache access separate so an average is taken. At the start of the trace the timer is stopped and at the end, after a million memory accesses it is stopped again. This way there is enough difference to differentiate between configurations.

In order for the timing test to only contain the cache access time and not anything else the tracefile is read to memory beforehand. Before any timing tests start a memory block is allocated for the trace to be read to. This way the disk access, which takes a lot of time, is not included in the timing test. The addresses are now directly available in memory.

The results of the timing tests are dependent on the processor used and what other processes might still be running. All the tests will be performed on the same processor and all other processes running will be, as far as possible, terminated for a fair comparison.

Worst case

In the timing test the same memory access patterns are used as in the hit ratio test. This gives for a realistic representation from the real application but does make it tricky to compare the actual timing. Since the cache is updated after a memory call the overhead timing is also dependent on the hit/miss ratio. If all cache calls would be misses the associative caches would have longer overhead timings.

In order to compare the overhead timing without the hit ratio a worst case test is performed. Here the same memory access pattern from the tracefiles is followed but no data is written to the cache. So each memory access the cache has to check if the data is available, but after the call the cache is not updated. This way every cache call will be a miss and the worst case overhead timing can be determined. The way this timing is measured is the same as in the overhead timing test in section 6.2.2.

Execution times

The next test will be the comparison of estimated execution times. This will be a way to simulate the actual data transfer times in a real application. In this test the cost of a cache miss will be included. The test will be based on the hit/miss ratio test in section 6.2.2 and the timing test in section 6.2.2 but with added cache miss costs. Introducing the miss penalties makes this test not only application but also platform specific.

The way these costs are introduced is with a fixed data transfers speed based on measurements from other research. The test will follow the timing test but also keeps track of the number of hits and thus indirectly also the number of misses. At the end of the test the number of misses is multiplied with the data size and a transfer speed to determine the extra timing costs for the misses. This is done separately for each cache, since the data size differs. The extra data transfer times are added to the overhead timing to find approximate overall execution times.

Other research done about exploiting FPGA platforms for PLF calculations specifically with RAxML provides great insights in typical transfers speeds [33]. Here accelerator cards are used to speed up the RAxML program and timing measurements are done. The measurements used are not listed in the published paper but contact with the authors provided useful data. Transferring data from the host to the accelerator can be done at 9.85GB/s. Transferring data back from the accelerator to the host is done at 12.81GB/s. In real world applications data transfer speed is not constant but for this test that assumption is made. This will be the last test in the state space reduction.

With these transfer speeds the costs for cache misses can be determined for each tracefile and node type. In the table 6.2.1 the extra costs for each cache miss is given.

Table 6.2.1: Miss penalties for each tracefile based on the data in [33]

	Bytes per tip node	**Bytes per inner node**	**Miss penalty tip node [µs]**	**Miss penalty inner node [µs]**
100 sequences	536	68608	0.0544	6.9653
250 sequences	548	70144	0.0556	7.1212
500 sequences	543	69504	0.0551	7.0562
1000 sequences	725	92800	0.0736	9.4213

the data transfer back to the host memory is not dependent on the hit/miss ratio since a write through policy is used. Every calculated value is returned back to the host memory after using. The result of a calculation on the accelerator is always in the form of an inner node data block. With the assumed transfer speed from the paper, the trace length of one million accelerator calls and the data size known, the costs for data transfer from accelerator to host can be determined. In table 6.2.2 transfer costs for the 1000 sequence trace is shown. Here the for each of the inputs and for the output the transfer costs are shown, based on a non

Table 6.2.2: Transfer costs for a trace of 1000 nodes.

	Number of inputs	**Node size [bytes]**	**Total size [GB]**	**Transfer rate[GB/s]**	**Transfer time[s]**
Tip input	729799	725	0.5	9.85	0.05
Inner input	1270201	92800	117.9	9.85	11.97
Output	1000000	92800	92.8	12.81	7.24

Table 6.2.3: Transfer times for data from accelerator to host per tracefile.

	Transfer Tip input [s]	**Transfer Inner input [s]**	**Transfer output [s]**	**Total transfer time[s]**
100 sequences	0.04	9.27	5.36	14.67
250 sequences	0.04	9.58	5.48	15.09
500 sequences	0.04	9.40	5.43	14.86
1000 sequences	0.05	11.97	7.24	19.27

cached situation, so the worst case transfer times. The same can be done for the other tracefiles resulting in the total worst case transfer times shown in table 6.2.3

6.2.3 Application performance using cache

Up until this point all tests are focused on the data transfer part only. For the final performance test it is interesting to look at the bigger part of the accelerator use. In this test the effectiveness off the cache is combined with a model for the datapath. This way the memory bottleneck is put into perspective and the performance of the entire application is shown.

This is achieved by including a model for the datapath, based on the latency, throughput and clock frequency of the hardware kernel running on the accelerator. The times that follow from this data model are included in the previous test to find a total execution time for both the data transfer and datapath.

In this test the best performing cache configurations from the previous tests are compared to a benchmark without a cache and a realistic performance indication can be given. To determine the computation time eq. (6.2.1) is used.

$$\text{Total computation time} = \frac{\text{Number of sites per sequence}}{\text{Sites per seconds}} * \text{Number of calculations} \tag{6.2.1}$$

The values used for the data model also come from the paper mentioned for the execution test [33]. Here the algorithm produces 16 double-precision values every two cycles at a clock frequency of 222MHz. This resulted in a measured throughput for testing DNA data on AWS at 96968756 alignment patterns per seconds. Each of the traces have a different number of alignment patterns which is the same as the number of bytes required for a tip node. With a length of one million accelerator calls in each tracefile this results in the computations times shown in table 6.2.4

6.2.4 Test reliability

There are some notes to be made about the reliability and applicability of the tests. Some simplifications are made. The first one to mention is the tracefiles extracted from RAxML. Since RAxML exhibits partly non deterministic behavior, no two traces for a single set of DNA

Table 6.2.4: Total computation times per tracefile.

	Number of patterns	**Computation time [s]**
100 sequences	536	5.53
250 sequences	548	5.65
500 sequences	543	5.60
1000 sequences	725	7.48

sequences are the same. For better results average over multiple runs would be better but less feasible in terms of computation times for all the tests.

A bigger simplification is the fact that in initial tests the hits and misses for both caches are weighted equally in the total score. The extra penalty for misses in the inner cache is introduced at the execution times test.

Finally, the cache size are fixed to the same number of cachelines because both have a similar number of data entries. However a cacheline for the inner node cache is 128 times larger than a cacheline for the tip node cache. In practice the application might benefit with a much larger tip cache at the expense of a small portion of the inner node cache. But the the misses from the tip node cache are not as costly so further research must be done in the optimization of this idea.

Chapter 7

Results

In this section the results for the different tests described in chapter 6 are shown and elaborated upon. These results give a clear indication of the possible performance increase in a real world application. As mentioned, the performance of the cache is evaluated for each characteristic separately. In section 7.1 the different replacement policies are compared, in section 7.2 the impact of the level of associativity is shown. The last characteristic that is evaluated is the cache size in section 7.3. An overall application performance comparison is done in section 7.4. Finally, in section 7.5 the ease of use in an OpenCL application is shown.

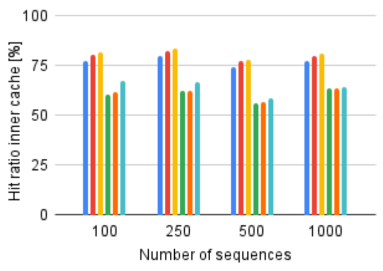
7.1 Replacement policies comparison

To compare the different replacement policies the first test is the hit/miss ratio comparison. Since both implemented caches operate separately and do not influence the performance of the other, the performance of each cache is evaluated separately. All tests results that are shown are with cache sizes of 8 cachelines. This is the smallest number of chachelines where there is still a distinction in behavior between all four different configurations. For a cache with 4 cachelines the fully associative and 4-way associative would have te same behavior. The tests have also been performed for sizes of 4, 16 and 32 cachelines. All show similar results as shown in the graphs below unless explicitly stated otherwise.

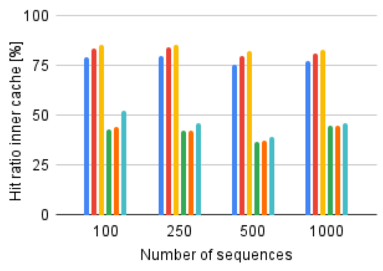
7.1.1 Hit/miss ratio

In fig. 7.1.1 the performance of the different replacement policies for the caches can be seen in terms of hit/miss ratio.

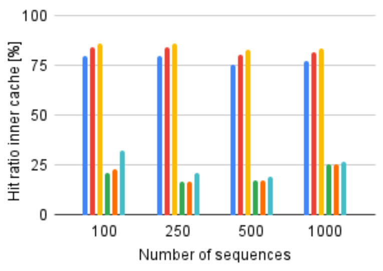
For the inner cache it can clearly be seen that the Random, FIFO and LRU replacement policies outperform the other three. This effect gets larger for higher level of associativity. The LFU and MFU replacement policy do not perform well for small cache sizes, 8 cachelines in this test. There is not enough differentiation between the cachelines to determine the best line to replace. The low scoring for MRU can be explained by the type of memory access pattern that RAxML uses. MRU performs best in a cyclical access pattern which RAxML does not have. As for the tip cache, the results slightly differ from the inner cache replacement policy. Once again Random, FIFO and LRU perform well, but this time MFU also performs well. For a higher number of nodes or a more associative cache however the performance drops. The performance of the MFU is greatly improved by a lower number of nodes per cacheline. In fig. 7.1.2 the performance for a cache with 32 cachelines is shown, here MFU actually performs best. In the real application however the number of nodes will most likely be larger than 1000



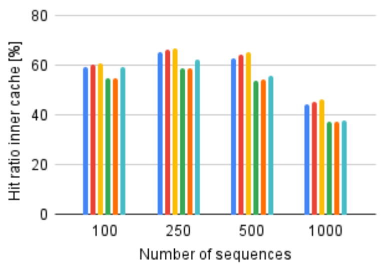
(a) 2-Way associative inner cache.



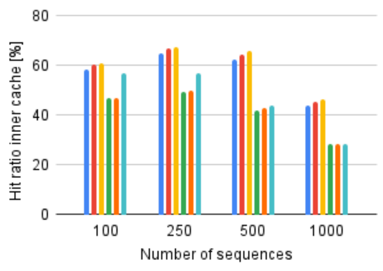
(b) 4-Way associative inner cache.



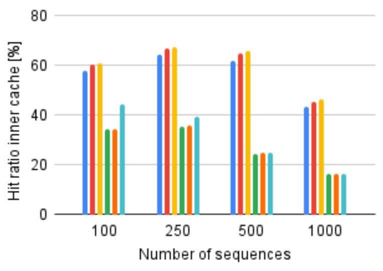
(c) Fully associative inner cache.



(d) 2-Way associative tip cache.



(e) 4-Way associative tip cache.



(f) Fully associative tip cache.

🟦 Random 🟥 FIFO 🟨 LRU 🟩 MRU 🟧 LFU 🟦 MFU

Figure 7.1.1: Hit/miss ratio comparison of replacement policies for caches configured with 8 cachelines.

and the number of cachelines will be very small. So even though MFU scores well in some instances it most likely will not perform as well in a real application.

7.1.2 Overhead timing

Next comparison metric is the overhead timing. The differences in timing for the leftover replacement policies for both caches are very small. There seems no clear differentiation for the timing costs of the different replacement policies. In fig. 7.1.3a the comparison of timing costs for a fully associative inner cache can be seen, a fully associative cache has the most complex overhead and thus shows most difference in the comparison of replacement policies. Other configurations show similar trends but scaled down slightly. Due to this lack of differentiation between configurations only the results of the fully associative caches are shown for the rest of the replacement policy tests. There is one outlier here for the trace with 500 nodes and the LRU or MRU replacement policy. This outlier was not present in other cache sizes or configuration.

The same holds true for the timing comparison for the tip cache. In fig. 7.1.3b the timing cost comparison for a fully associative tip cache can be seen. Here the difference is even smaller and no outlier is present. Similar results are present for different configurations and cache sizes. All differences in results per trace are within 1ms margin which is negligible when compared to the impact the hit/miss ratio has on the total transfer time.

7.1.3 Worst case timing

In section 7.1.3 the worst case execution times of the different replacement policies for the inner cache can be seen. Here is a clear differentiation for the different replacement policies. For higher associative the LRU and LFU have triple the overhead timing costs as the other policies. For the tip cache this difference is not present, see section 7.1.3. Here the Random replacement policy actually scores slightly worse for a smaller number of nodes.

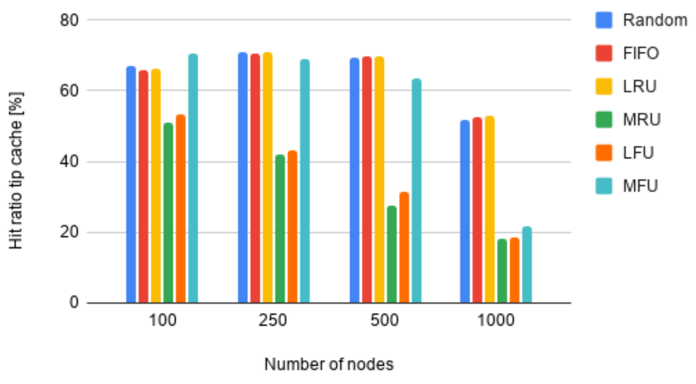
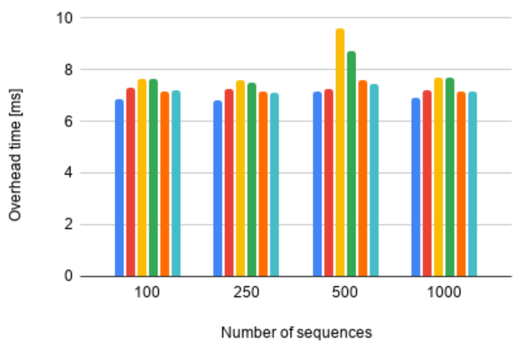
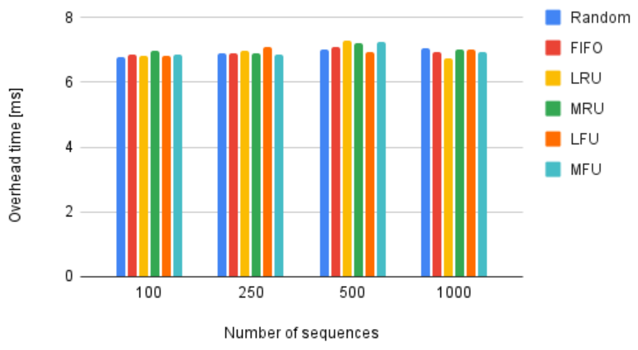


Figure 7.1.2: Hit ratio for a fully associative tip cache with 32 cachelines.



(a) Fully associative inner cache.



(b) Fully associative tip cache.

Figure 7.1.3: Overhead timing replacement policy comparison for caches configured with 8 cachelines.

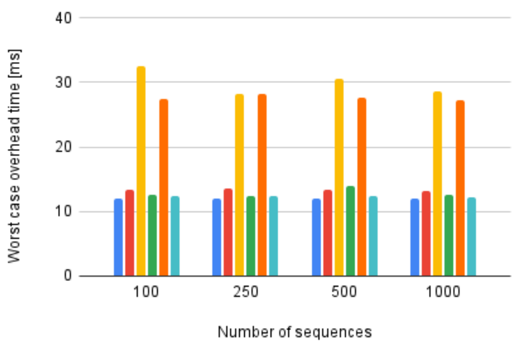
For both cases the differences are still negligible in the total performance picture. In section 6.2.2 it is shown that the total worst case transfer time for all the tracefiles is in the range of roughly 15-20 seconds. A difference in overhead timing of 20µs is not going to influence the total noticeably in comparison to a difference in hit/miss ratio.

7.1.4 Execution times

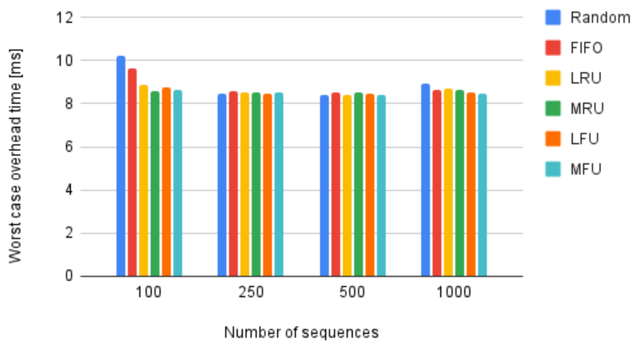
A final comparison for the replacement policies is done on the execution times. Here the overhead timing results are combined with the cache miss penalty. In section 6.2.2 the used penalties are calculated and described. In fig. 7.1.5 the different remaining replacement policies for both caches can be seen. Since the overhead timing has a negligible impact on the total transfer time the graph essentially shows the same results as in the hit/miss ratio test in section 7.1.1.

7.1.5 Summary

From the previous tests a clear difference among the replacement policies in terms of performance can be seen. Even though LRU has the highest overhead and worst case timing in the execution time test it still performs best. This shows that the hit/miss ratio in this case is much more influential than the overhead timing. From these results Random, FIFO and LRU came up as the best performing replacement policies. MRU, LFU and MFU all perform comparable to each other. These replacement policies typically perform better in cyclical memory access patterns as described in section 2.3.4. Looking at the tracefiles from RAxML it can be seen that

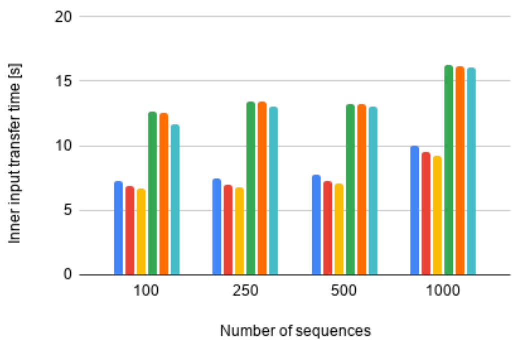


(a) Fully associative inner cache.

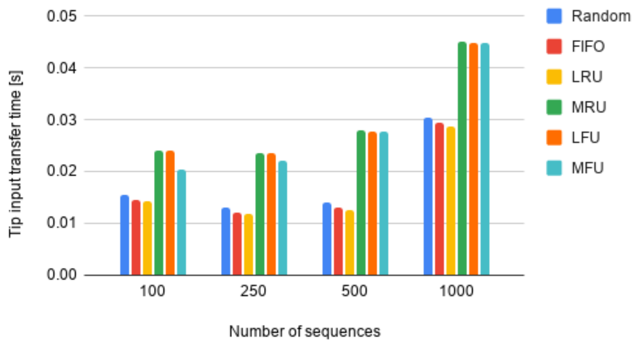


(b) Fully associative tip cache.

Figure 7.1.4: Worst case overhead timing replacement policy comparison for caches configured with 8 cachelines.



(a) Fully associative inner cache.



(b) Fully associative tip cache.

Figure 7.1.5: Total execution time replacement policy comparison for caches configured with 8 cachelines.

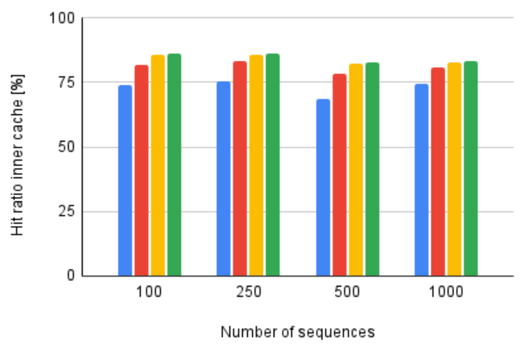
it does not contain such a cyclical access pattern making it not surprising that these replacement policies do not perform as well. However even in the worst case there is still it hit/miss ratio of more than 15% so overall performance is still better than no cache at all.

7.2 Level of associativity

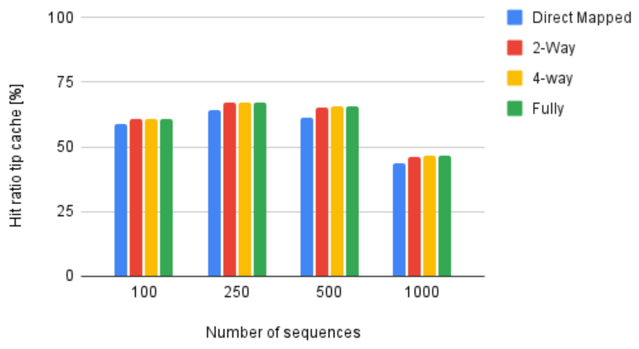
The next characteristic to look at is the level of associativity. A similar approach as for the replacement policies is chosen for this comparison. First a hit/miss ratio comparison is done. Then the overhead and worst case overhead timings are compared. Finally, the total execution times are compared. To keep the testing structured not all tests performed are shown in this section. All results shown are cache with the LRU replacement policy. Since that was the best performing policy it is used here. All tests are executed with all different replacement policies and show similar behavior as the presented LRU caches unless explicitly stated otherwise.

7.2.1 Hit/miss inner

The first test is the hit/miss ratio. The hit/miss ratio comparison can for the inner and tip cache can be seen in fig. 7.2.1a and fig. 7.2.1b respectively. For the inner cache the performance increases with the level of associativity but the differences between any of the associative caches are small. Only the direct mapped cache stands out in performance. The tip cache shows very



(a) Inner cache.



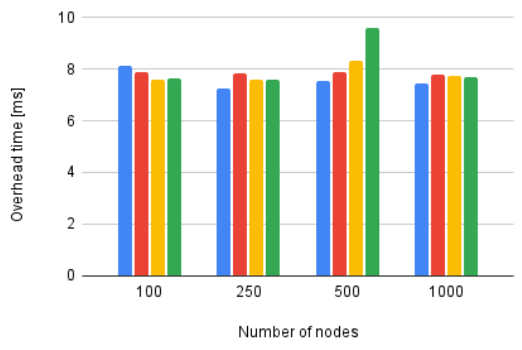
(b) Tip cache.

Figure 7.2.1: Cache associativity comparison based on hit/miss ratio for caches configured with 8 cachelines and LRU.

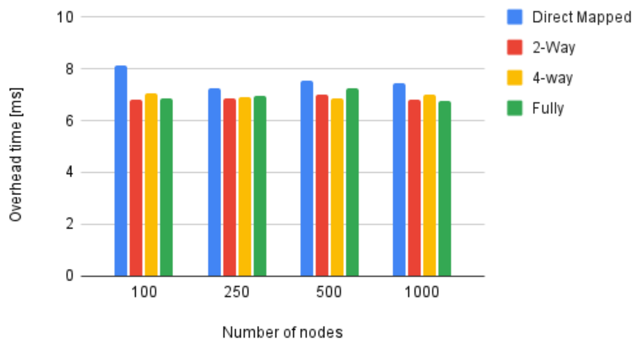
little influence of the level of associativity on the hit/miss ratio. Only the direct mapped cache slightly under performs in comparison to the others. The other replacement policies show a similar result.

7.2.2 Overhead timing

The level of associativity comparison based on the overhead timings shows just as little differentiation as the hit/miss comparison. In fig. 7.2.2a the difference for an inner cache with LRU is shown. Here is one outlier which is the fully associative cache for a data set of 500 sequences. Most likely this test has a higher number of misses which influence the performance of the fully associative cache the most. For the other replacement policies and cache sizes this outlier is not present the mutual differences are as small as all other differences shown in the graph. In fig. 7.2.2b the comparison for the tip cache with LRU is shown. Here is also very little differentiation between the configurations. All other replacement policies and cache sizes show similar results. Just as with the overhead timing in the replacement policy comparison all differences in overhead timing are negligible compared to the influence of the hit/miss ratio.



(a) Inner cache with LRU.

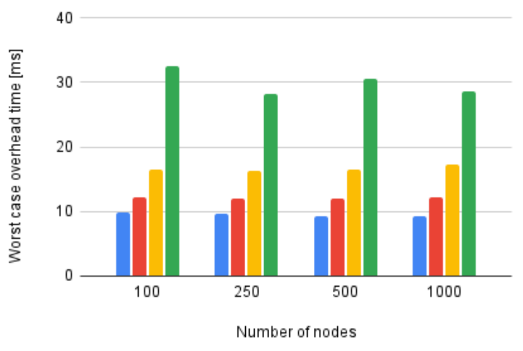


(b) Tip cache with LRU

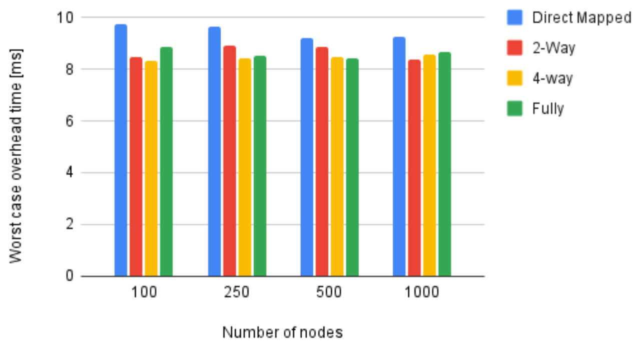
Figure 7.2.2: Cache associativity comparison based on overhead timing for caches configured with 8 cachelines and LRU.

7.2.3 Worst case timing

The worst case timing tests for different levels of associativity show more differentiation than the overhead timing test. There is quite a significant difference in overhead timing for higher



(a) Inner cache with LRU.



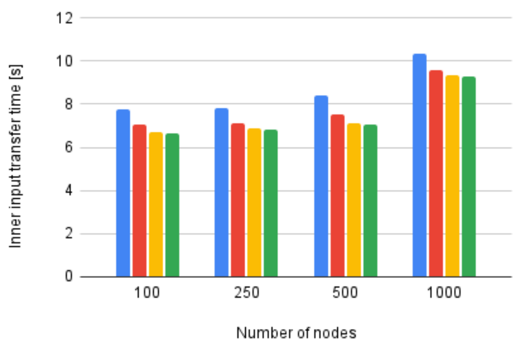
(b) Tip cache with LRU

Figure 7.2.3: Cache associativity comparison based on worst case timing for caches configured with 8 cachelines and LRU.

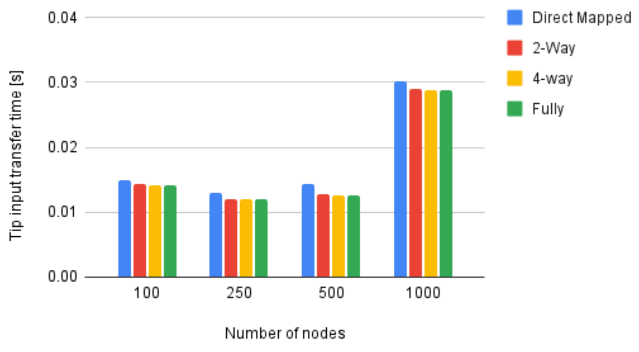
levels of associativity for the inner cache, as can be seen in fig. 7.2.3a. Here the differences in worst case timings are comparable to the difference seen in the replacement policies test in section 7.1.3. For the tip cache these differences are small as can be seen in fig. 7.2.3b. Here the direct mapped cache actually performs worst. All observed differences still fall within the range of difference that is negligible for the total transfer time.

7.2.4 Execution times

The last test is the total execution time comparison. Here once again the values mentioned in section 6.2.2 are used to calculate the costs of a miss penalty, The results for the inner cache and the tip cache can be seen in fig. 7.2.4a and fig. 7.2.4b respectively. Again, since the overhead timing is negligible on the total performance the graphs show similar results as for the hit/miss ratio. Where a higher level of associativity, especially for the inner cache, reduces the execution time more.



(a) Inner cache with LRU.



(b) Tip cache with LRU.

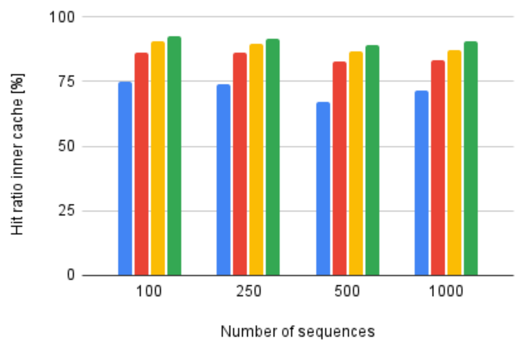
Figure 7.2.4: Cache associativity comparison based on total execution times for caches configured with 8 cachelines and LRU.

7.2.5 Summary

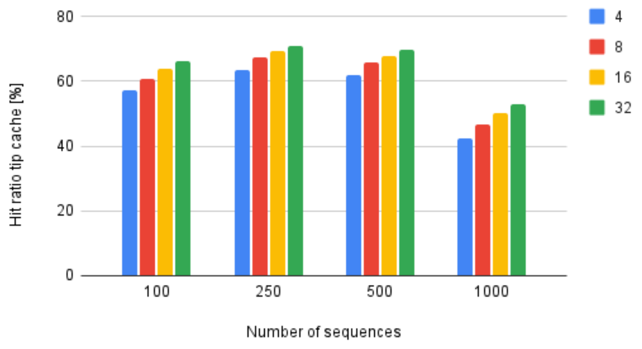
From the tests it can be said that for this application a higher level of associativity increases the performance. Even though this results in more overhead time costs, the reduced number of data movements weights heavier on the overall performance.

7.3 Cache size

As mentioned in section 6.2.2 the cache size most likely will be dictated by the available memory on the accelerator card. Since the previous tests show that the overhead timing adds negligible costs compared to the hit/miss ratio it seems logical that a higher number of cachelines will result in a higher increase in performance. To check if indeed the performance increases with the number of cachelines a test is performed. In this case only the hit/miss ratio test since most performance difference will come from that metric. In fig. 7.3.1 the hit/miss ratio for the inner and tip cache can be seen. Both caches are configured as a fully associative cache with the LRU replacement policy. The bars show different number of cachelines.



(a) Fully associative inner cache with LRU.



(b) Fully associative tip cache with LRU.

Figure 7.3.1: Configuration comparison for caches configured with 8 cachelines. Total execution time of the data transfers.

As expected the larger cache size have a higher hit/miss ratio and perform better.

7.4 Application performance using cache

For the final test it is interesting to see how the different configurations perform compared to each other in the larger picture of the accelerator use case. So the final test will have the four configurations all with the LRU replacement policy since it is clearly performing best. An overview of the different combinations can be seen in table 7.4.1.

Table 7.4.1: Different combinations of settings used in final testing.

	Inner cache configuration	**Inner cache replacement policy**	**Tip cache configuration**	**Tip cache replacement policy**
Combination 1	Direct mapped	n/a	Direct mapped	n/a
Combination 2	2-Way associative	LRU	2-Way associative	LRU
Combination 3	4-Way associative	LRU	4-Way associative	LRU
Combination 4	Fully associative	LRU	Fully associative	LRU

In this test the data model is included to put the transfer times into perspective of the entire execution times. The data model used is explained in section 6.2.3 and is based on measured values from previous research. In fig. 7.4.1 the different selected combinations in table 7.4.1 and a no cache implementation are compared for the tracefile with a 1000 nodes. This is done separately for the inputs, tip and inner, the processing and the output times. The processing and output times in this case are all constant.

The major difference is with the inner input. Here the transfer times is reduced from 12s to 2s in the best scenario. Another thing that can be noticed is the small impact in general the tip input has on the total execution time. In the graph the bar for the tip input is shown but barely visible for the cached combinations.

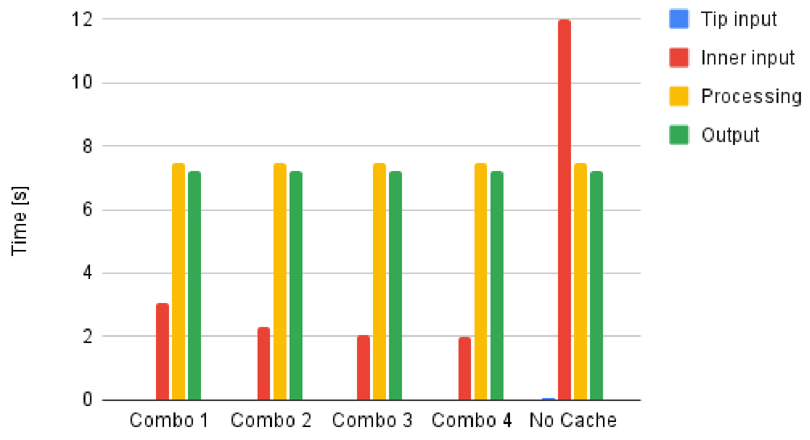


Figure 7.4.1: Comparison of the different combinations.

In fig. 7.4.2 the total execution time for each trace can be seen. In this case a new factor is included, namely the matrices transfer. RAxML required a 144 value double-precision matrix for each calculation. These values have to be transferred to the accelerator but this is a constant value so during the rest of the testing this is not included. In the end the extra transfer time added by this matrix transfer is very small. One million times a transfer of 144 double-precision values results in a total of 1.2GB, at a transfer rate of 9.85GB/s this only adds 0.12 seconds to the total execution time. This extra transfer does not influence the total by much but does make sure that the result is representative of a real application.

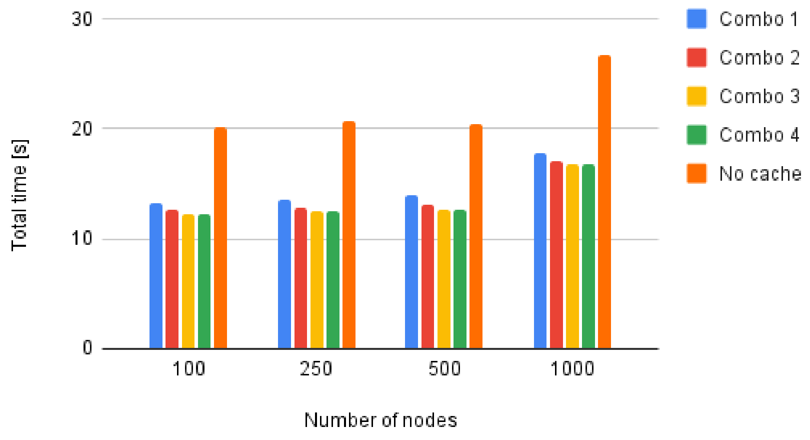


Figure 7.4.2: Comparison of the total execution times per trace.

All the chosen cache configurations perform very well compared to no caching solution. The difference between the combinations in this case is relatively small but in larger use cases it could add up to a significant difference.

7.5 Demonstration

A last test will be a cross platform execution. The goal of this test is to show the ease of use of the software cache and not the different performance as in previous tests. Here the transfer from the pure C code to an inclusion of OpenCL will be made. The entire code is made compatible with OpenCL and a simple vector add kernel will be used to replicate the behavior of an accelerator card. This is highly simplified in comparison with the RAxML code but can give a good indication about cross platform execution.

The test will be executed on the combination of a CPU and an internal GPU on a laptop. This test cannot say anything about the final performance of the cache implementation but purely shows the ease of use.

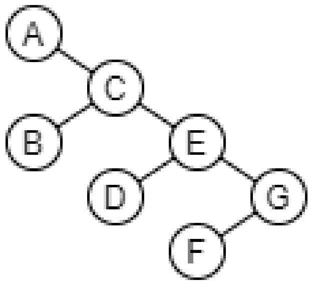


Figure 7.5.1: Tree structure used to test the OpenCL implementation

For this test a simple vector add example code for OpenCL is used as a base [34]. A small very small tree structure is made, fig. 7.5.1, where every node represents a vector of integers. The algorithm in this case is an addition of the two underlying childnodes. In this tree the nodes *A, B, D* and *F* are the tip nodes and are assigned with random data, *C, E* and *G* are empty vectors to store the intermediate data. The final node, *G*, should contain the sum of all tip nodes at the end of the calculations. This way the correctness of the cache implementation can be verified.

7.5.1 Code difference

To show the ease of use code snippets are given with and without the cache. Most of the code remains the same and is not shown in this section, only the differences are highlighted.

Cache instantiation

In the beginning of the code the cache needs to be instantiated. This is achieved by calling the `CreateCache` function and can be seen in listing 7.1. No code needs to be removed at this stage and no OpenCL function has to be replaced.

Listing 7.1: Code snippet of cache instantiation.

```
    struct Cache_t* myCache = CreateCache(8, dataSize, 32, direct_mapped
        , fifo_RP);
```

Data from host to accelerator

For the data transfer from host to accelerator the example uses the code shown in listing 7.2 to create `cl_mem` objects and assign the correct data to it before providing it as argument to the kernel.

Listing 7.2: Code snippet of data transfers without cache

```
//Create the input arrays in device memory and copy the host
    pointers
input1 = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input1, &err);
checkError(err, "Creating buffer input1");
input2 = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input2, &err);
checkError(err, "Creating buffer input2");
output = clCreateBuffer(context, CL_MEM_READ_WRITE, dataSize, NULL,
    &err);
checkError(err, "Creating buffer output");

// Set the arguments to the compute kernel
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &input1);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &input2);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &output);
checkError(err, "Setting kernel arguments");
```

To implement the cache the code has to be changed to the code shown in listing 7.3. The code is very similar but there are two key differences here worth mentioning. First it is important for defining the output that the `CL_MEM_COPY_HOST_PTR` argument is not passed in the function. This will make sure that the output will always be written to the cache even if an outdated version of that entry is already present. The second thing is that for the output instead of a `NULL` pointer now the pointer pointing the data in the host memory space is given. This way the cache controller can determine the location of the output in the cache memory and can keep track of the entries in the cache. The only other difference with the original OpenCL function arguments is the passing of the pointer of the cache.

Listing 7.3: Code snippet of data transfers with cache

```
//Write to the cache on the device memory
input1 = clCreateCacheBuffer(context, CL_MEM_READ_WRITE |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input1, &err, myCache);
checkError(err, "Creating buffer input1");
input2 = clCreateCacheBuffer(context, CL_MEM_READ_WRITE |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input2, &err, myCache);
checkError(err, "Creating buffer input2");
output = clCreateCacheBuffer(context, CL_MEM_READ_WRITE, dataSize,
    h_output, &err, myCache);
checkError(err, "Creating buffer output");

// Set the arguments to the compute kernel
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &input1);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &input2);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &output);
checkError(err, "Setting kernel arguments");
```

Note that the error checking functionality built in OpenCL is still functional with the added cache functionality. In both examples the OpenCL defined error type is provided as an argument and can be used to check correctness.

Data from accelerator to host

After the kernel is executed the output data from the accelerator can be transferred back to the host. The original code uses a `clEnqueueReadBuffer` function to achieve this which can be seen in listing 7.4

Listing 7.4: Code snippet of retrieving data from accelerator.

```
    // Read back the result from the compute device
    err = clEnqueueReadBuffer(commands, output, CL_TRUE, 0, dataSize,
        h_c, 0, NULL, NULL);
    checkError(err, "Reading back h_c");
```

With the cache implementation this method can still be used. This does however not update the access order in the cache for replacement tracking so performance might drop due to improper tracking of usage for replacement but correctness is still insured. The caching software does however also offer some extra functionality in the form of the `clEnqueueReadCacheBuffer` function. The use of this function can be seen in listing 7.5.

Listing 7.5: Code snippet of retrieving data from cache.

```
    // Read back the result from the compute device
    err = clEnqueueReadCacheBuffer(commands, NULL, CL_TRUE, 0, dataSize,
        h_c, 0, NULL, NULL, myCache);
    checkError(err, "Reading back h_c");
```

This function allows for data other than the most recent result to be retrieved from the cache memory. For this function no location on the device memory has to be given as an argument since the cache controller contains that information. Only the location in the host memory where the data is to be stored is required. The controller will find the location in cache and if the requested data is not in cache the function will return `1` which can be used with the error functionality to identify such an occurrence.

In the case of this example it is possible to retrieve the intermediate data of nodes *C* and *E* after all three computations are completed. If of course the cache size is big enough and the data is not yet overwritten with new data.

Cache destruction

The last piece of code that is added to the example is the cache destructor function `FreeCache`. At the end of the example listing 7.6 is added.

Listing 7.6: Code snippet of cache destruction.

```
    FreeCache(myCache);
```

These are all the changes required to use the cache within this simple example.

7.5.2 Performance

Now that is shown how little the code has changed with a cache included it is time to see how the execution is affected. This is a very simplified example but does show the correctness and potential of the caching software. In table 7.5.1 the data transfer order for the regular case is shown. In this case there are 6 transfers from host to accelerator and 3 transfers back from the accelerator to the host.

Table 7.5.1: The data transfer order for the nodes without a cache.

Nodes	{A,B}	C	{C,D}	E	{E,F}	G
Direction	h→ d	d→ h	h→ d	d→ h	h→ d	d→ h

When taken a large enough cache size the expected data transfer order can be seen in table 7.5.2. Here the data from nodes *C* and *E* are stored in the cache and never transferred. This case would require 4 transfers from host to accelerator and 1 transfer back.

Table 7.5.2: The data transfer order for the nodes with a cache.

Nodes	{A,B}	D	E	G
Direction	h→ d	h→ d	h→ d	d→ h

For the test the cache size is set to 8 cachelines meaning that there should be enough space for all nodes and result in the optimal data transfers as described above. The results show that indeed in this case only 4 data transfers from host to accelerator are done and only a single data transfer back to the host while maintaining the correct result for the additions.

This shows that with very little change to existing OpenCL code performance of algorithms can be improved with software caching.

Chapter 8

Discussion

In this chapter different parts of the project are discussed. First the overall design and the possible implications or limitations of the made design choices in section 8.1. In section 8.2 the experimental setup and the limitations of that setup are discussed. Next the actual results of the testing are discussed in section 8.3. Finally, in section 8.4 some thoughts about the expendability to other applications are given.

8.1 Design

During the design process some design decisions were made to keep the project feasible and still prove that the concept of software caching for accelerator cards can be beneficial. The first of these decisions is complete programming in `C`. Currently that is the only language supported but OpenCL does support for instance `C++` syntax or Python. The code is not overly complex and does not use functions that are not supported by either of these two languages. This means that the code should be relatively simple to be converted to `C++` or Python for support of a wider spectrum of applications.

The current design only supports three different associative cache configurations; 2-way, 4-way and fully associative. For larger cache sizes the difference between 4-way and fully associative is very large and larger set associative cache configurations can simply be implemented. The restriction that set sizes and number of cachelines in general have to be powers of two originated from the goal of lowest possible overhead costs. During testing the overhead timing appeared negligible compared to the influence of the hit/miss ratio. This could mean that the extra overhead costs for a more complex controller that allows for an arbitrary number of cachelines or set size, are compensated by a higher hit/miss ratio. This way more of the available memory space on the accelerator can be used for caching. Further testing is required to confirm this hypothesis.

Currently only a limited amount of replacement policies are included in the design. Many more replacement policies exist and for a software cache the implementation is simpler than for a hardware implementation. The current supported replacement policies were chosen based on the level simplicity, both in functionality and implementation, that these policies have. Other policies are not tested but perhaps show very different results. In the end it is mainly dependent on the type of application used.

8.2 Experimental setup

The phylogenetic data used during testing is generated with tools that represent actual data. This is a solid base for the testing. For the testing the memory access trace extracted from RAxML is limited to one million memory calls. In a real application this would be much larger, especially for higher number of sequences. Since RAxML operates on subtrees this reduction of the trace still provides a representative memory access trace. The total number of cache hits and misses would increase in a real application but the ratio will probably not. Another difference with the generated phylogenetic data is the number of alignment sites and thus the data size. In real world application this could easily be a factor 1000 larger, increasing the data size per node. This would mean that the execution times from the tests are not representative but the ratios still are. In absolute terms the caching software will show more effect in these real scenarios.

All development and testing has been performed on a Windows device. This means that for the timing experiments the results differed from time to time. Windows does not allow full control of background processes and it was clear that some outliers were present. To solve this an average over multiple runs has been taken but a better solution would be to run the tests on a Linux device. In a Linux environment timing tests should show similar results in different runs. The differences in timing of the current testing can still be used to show the negligible effect of the overhead timing compared to the hit/miss ratio.

Unfortunately testing on an actual accelerator card, directly or via AWS, was not possible in the time frame of this research. This means that no timing results from an actual application can be shown. However the models provide a solid base to show the proof of concept.

8.3 Results

Many tests have been performed and a lot of results have come out of these. All these results show the performance increase the caching software can provide but only in the context of phylogenetics right now. The performance of the different replacement policies was very noticeable. Since RAxML does not have a cyclical memory access pattern the MRU, LFU and MFU replacement policies performed considerably lower than the other policies. This does not mean that the policies itself are inferior, but they are not suitable for this application. One interesting thing to note is the performance of the Random replacement policies. In essence, it is one of the simplest replacement policy one can imagine but the performance was in a similar range as the FIFO and LRU policies. It does result in non-deterministic behavior which might be undesirable in certain applications that can use caching.

From the testing it became apparent that the overhead timing costs in any combinations of settings was negligible compared to the impact of reducing the data transfers. One of the main downsides in a hardware cache for an associative cache are the added costs in the form of hardware and time. For a software cache this extra hardware costs do not exist and the extra time costs are well compensated by the increased performance. In almost all tests the higher associative caches performed better. So a better performing cache with no real added costs would make the choice for a higher level of associativity very simple. This also allows for extra flexibility with the replacement policies that are not present on a direct mapped cache.

In all tests the sizes for both caches were the same to reduce the state space of possible combinations of settings. For an actual implementation this generalization is not required and it might be better to have different size caches for different types of nodes. The current data does not give any insight on this hypothesis and further testing is required.

8.4 Other applications

The results show that indeed for an accelerated phylogenetic likelihood function the cache improves performance. The design of the cache however facilitates the straightforward deployment with other applications. Most accelerated applications using OpenCL, currently only in C syntax, could potentially benefit from this software cache. The implementation into an existing design is straightforward as shown in section 7.5. The goal of this research was to evaluate performance on tree-based algorithms because of locality of data within a tree structure. This does not mean that it cannot improve performance of other memory bottlenecked accelerated algorithms. Applications that have repeated memory accesses potentially could benefit from this software cache. The code essentially can be implemented in any OpenCL application but the performance and best settings will heavily depend on the application and further testing is required for each type of application.

Chapter 9

Conclusion

In this chapter the conclusions of this research are discussed. First the set research questions are answered and after that possible future work is discussed.

9.1 Research questions

Before answering the main research question, first the subquestions are answered in the order as presented in section 3.1, starting with the subquestion:

> *Can standard caching techniques be used for tree-based algorithms?*

During this research a software cache was designed based on the OpenCL standard. The design supports standard caching techniques found in hardware based cache designs, namely indexing, inclusion of associativity and support for different replacement policies. With the tests performed on the PLF tree-based algorithm it shows that the software can indeed be used to alleviate the number of data transfers. Currently the testing for tree-based algorithms is limited to the PLF algorithm, so no conclusive answer can be given towards tree-based algorithms in general. But since the design is OpenCL based, it is easily implementable in existing code and it is highly likely that other tree-based algorithms can benefit from this caching software. In fact in section 7.5 it is shown that even from origin non tree-based algorithms such as the vector addition algorithm can be extended to use the designed software cache. Based on this research no conclusions can be given on the performance in other applications however it can be concluded that standard caching techniques can be used on algorithms besides tree-based algorithms.

> *What is the effect of caching on data movement and accelerator performance for the phylogenetic maximum likelihood function?*

The results show very promising performance increases for the PLF algorithm. With the generated data, the data transfer model and the computational model it is shown that the number of data transfers from the host to the accelerator can be reduced up to 90% in the best case scenarios which reduces the total transfer times to a sixth of the original transfer times in section 7.4. The overall accelerator computation time, based on the models used, has been decreased between 31.6% and 39.9% for different tree sizes. This shows high potential for the use of this cache within the RAxML software for larger scale computations.

What cache characteristics enable tree-based algorithms to benefit from caching when accelerator cards are deployed?

The final subquestion is about the different cache settings used. The tests only covered the PLF so no generic statements can be made on the performance of the different cache characteristics. However for the PLF some clear conclusions can be drawn. First, as expected, a larger cache results in better performance. The extra overhead timing required for a larger cache for look ups is negligible compared to the decrease in data transfers.

Next is the level of associativity of the cache. From all tests a higher associativity resulted in less data transfers. Again due to the negligible amount of extra overhead timing costs that comes with a higher level of associativity it always seems worth it. However the differences are not very large and even a direct mapped cache performs in a similar range as the associative cache configurations.

Finally the replacement policies. During testing it became apparent that the choice of replacement policies is very important since it influences the performance greatly. It will be depended on the type of applications which replacement policy performs best.

Main research question

With all the subquestions answered the main research question can now be answered.

How effective can caching be in alleviating the memory bottleneck when using an accelerator card for data-driven memory-bound tree-based algorithms?

In spite of the fact that no conclusive result can be given about performance of software caching for tree-based algorithms in general due to the fact that only the PLF was tested, still very positive results have shown that the memory bottleneck can be alleviated by caching. In fig. 7.4.1 it can be seen that without caching 74% of the total execution time is required for the data movement, where with caching only 59% of the total execution time is required for data movement. In these results only the data transfers from host to accelerator are targeted so when looking at the percentage of time required for data transfers from host to accelerator it is reduced from 44% of the total execution time without caching to just 12% with caching.

Certainly for the PLF algorithm it can be concluded that the use of caching reduces the effect of the memory bottleneck present in this application. It is likely that other tree-based algorithms can also benefit from caching, more research is required to get conclusive results for tree-based algorithms in general.

9.2 Future work

Based on the findings during this research some recommendations on further research are made.

9.2.1 Other applications

First is to expand upon this research and verify the performance of the caching software on other tree-based algorithms. Having multiple algorithms that benefit from caching shows the generic nature of the designed software and the principle behind it.

Besides the different algorithms it is also advised to perform some tests on different platforms. Again this will show the generic nature of the design. Currently the performance has been measured with models but not on actual accelerator devices. Until proper testing on hardware accelerator cards has been conducted it is difficult to give conclusive answers about the actual performance increase that is provided by the caching software.

9.2.2 Expanding cache functionality

As a proof of concept the current software offers the basic caching techniques from traditional hardware design. However there exists many more techniques that could be implemented to further enhance the functionality and usability of the caching software.

Configurations

One of the current limitations is the number of supported cache configurations and cache size. Originally the number of cachelines was limited to a power of two to reduce the overhead timing costs. During testing the overhead timing has shown negligible compared to the overall performance increase. Since the controller is purely software based it should be simple to change the indexing of the cachelines. Instead of using a number of bits of the memory address, the modulo of an integer representation of the memory address could be taken. In hardware this would be complicated due to the extra divisions added but in software this would be much simpler. This would allow for an arbitrary amount of cachelines to be used in any design, meaning that more of the available memory space on the accelerator card can be utilized.

This other method of indexing can also be extended to the set associative configurations. The number of sets as well as the number of ways within each set could be an arbitrary number. These additions would improve overall usability of the cache and most likely also the performance. Further testing with this indexing method is required to verify that the added overhead costs are compensated by the reduction in data transfers.

Replacement policies

A similar thing can be noted about the replacement policies. Currently the number of supported replacement policies is limited to the base versions of most policies. As also described in section 2.3.4 there are far more complex replacement policies, that are for instance time aware or can dynamically change replacement behavior. Since the software controller is software based it easier to implement such replacement policies compared to hardware based cache controller designs. Further research has to be done into the performance and also extra overhead costs for these replacement policies.

Bidirectional data movement reduction

The focus during this research was the reduction of data movements from the host to the accelerator. This still leaves opportunities for reducing the transfer times from the accelerator back to the host memory. Currently the design uses a write through policy, meaning that every output of the accelerator is, besides saved in cache, directly transferred back to the host memory. The alternative is a write back policy where the data is not transferred back until it will be replaced by new data. Potentially this could reduce the number of data movements from the accelerator back to the host. It will depend on the application whether the functional correctness will remain the same for a write back policy and further research is required into this.

Bibliography

[1] Martha A. Kim and Stephen A. Edwards. Computation vs. memory systems: Pinning down accelerator bottlenecks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6161 LNCS:86–98, 2012.

[2] Nikolaos Alachiotis, Simon A. Berger, and Alexandros Stamatakis. Efficient PC-FPGA communication over Gigabit Ethernet. *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, (Cit):1727–1734, 2010.

[3] Nihar Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *XRDS: Crossroads, The ACM Magazine for Students*, 5(3es):2, 1999.

[4] Jim Handy. *The cache memory book*. Kaufmann, Morgan, 1998.

[5] Tom Vancourt and Martin C. Herbordt. Chapter 2 Elements of High-Performance Reconfigurable Computing*, 1 2009.

[6] Martin C. Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with FPGA-based computing. *Computer*, 40(3):50–57, 2007.

[7] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69–76, 2008.

[8] B Cao, T Srikanthan, and C H Chang. Efficient reverse converters for four-moduli sets. 152(20045116):193–207, 2005.

[9] Duncan Buell, Tarek El-Ghazawi, Kris Gaj, and Volodymyr Kindratenko. Guest editors' introduction: High-performance reconfigurable computing. *Computer*, 40(3):23–27, 2007.

[10] Alexei V. Finkelstein and Oleg B. Ptitsyn. Lecture 15: Caching [PowerPoint slides]. *Retrieved from University of Washington, course CSE378*, 2002.

[11] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[12] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior: Is It $\sqrt{2}$? *Proceedings of the 3rd Conference on Computing Frontiers 2006, CF '06*, 2006:313–320, 2006.

[13] Roel J. Wieringa. *Design science methodology: For information systems and software engineering*. Springer-Verlag, Berlin Heidelberg, 2014.

[14] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Two Paradigms on Research Essay Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–79, 2004.

[15] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[16] Maung K. Sein, Ola Henfridsson, Sandeep Purao, Matti Rossi, and Rikard Lindgren. Action design research. *MIS Quarterly: Management Information Systems*, 35(1):37–56, 2011.

[17] Xuechao Wei, Yun Liang, and Jason Cong. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. *Proceedings - Design Automation Conference*, pages 0–5, 2019.

[18] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen Mei Hwu, and Deming Chen. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, pages 2–9, 2018.

[19] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu Wing Tai. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. *Proceedings - Design Automation Conference*, Part 12828:1–6, 2017.

[20] David A. Bader, Usman Roshan, and Alexandros Stamatakis. Computational Grand Challenges in Assembling the Tree of Life: Problems and Solutions. *Advances in Computers*, 68(06):127–176, 2006.

[21] Nikolaos Ch Alachiotis. *Algorithms and Computer Architectures for Evolutionary Bioinformatics*. PhD thesis, Techinal University of Munchen, 2012.

[22] Joseph Felsenstein. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution*, 17(6):368–376, 11 1981.

[23] Michael Bachmann, Handan Arkin, and Wolfhard Janke. Multicanonical study of coarse-grained off-lattice models for folding heteropolymers. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 71(3):1–11, 2005.

[24] Stefan Schnabel and Wolfhard Janke. Accelerating polymer simulation by means of tree data-structures and a parsimonious Metropolis algorithm. *Computer Physics Communications*, 256:107414, 11 2020.

[25] Huan Zhang, Si Si, and Cho-Jui Hsieh. GPU-acceleration for Large-scale Tree Boosting. *arXiv preprint arXiv:1706.08359*, 2017.

[26] Maxim Shepovalov and Venkatesh Akella. FPGA and GPU-based acceleration of ML workloads on Amazon cloud - A case study using gradient boosted decision tree library. *Integration*, 70:1–9, 1 2020.

[27] Kholoud Shata, Marwa K. Elteir, and Adel A. EL-Zoghabi. Optimized implementation of OpenCL kernels on FPGAs. *Journal of Systems Architecture*, 97:491–505, 8 2019.

[28] Xilinx. Adaptable accelerator cards for data centre workloads. *https://www.xilinx.com/products/boards-and-kits/alveo.html*, 2020.

[29] Xilinx. Vitis software. *https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html*, 2020.

[30] Richard R. Hudson. Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics*, 18(2):337–338, 2 2002.

[31] Andrew Rambaut and Nicholas C. Grassly. Seq-gen: An application for the monte carlo

simulation of dna sequence evolution along phylogenetic trees. *Bioinformatics*, 13(3):235–238, 6 1997.

[32] Alexandros Stamatakis. RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 5 2014.

[33] Pavlos Malakonakis, Andreas Brokalakis, Nikolaos Alachiotis, Evripides Sotiriades, and Apostolos Dollas. Exploring Modern FPGA Platforms for Faster Phylogeny Reconstruction with RAxML. In *Proceedings - IEEE 20th International Conference on Bioinformatics and Bioengineering, BIBE 2020*, pages 97–104. Institute of Electrical and Electronics Engineers Inc., 10 2020.

[34] Simon McIntosh-Smith and Tom Deakin. HandsOnOpenCL. *Github repository, https://github.com/HandsOnOpenCL*, 2019.