Comparing priority queues with support for priority updates at arbitrary indexes

Erik Granberg
Abstract

Comparing priority queues with support for priority updates at arbitrary indexes

Erik Granberg

The research software URDME makes use of a priority queue that has support for updating the priority of enqueued elements at arbitrary indexes. To achieve this URDME currently relies on a Binary Heap. In this thesis, we consider and evaluate other priority queues, with an awareness of spatial locality and cache performance, that could potentially replace the Binary Heap in URDME.

Three different priority queue implementations were considered, Binomial Heaps, Fibonacci Heaps, and d-ary Heaps. The three heaps were then compared with a weighted combination of literature review, analysis of spatial locality, and analysis using the RAM-model.

Due to a lot of the literature being critical of the practical performance of the Fibonacci Heap, as well as the poor spatial locality exhibited by both the Fibonacci Heap and Binomial Heap, these two were excluded from further evaluation. An instance of the d-ary Heap, the 4-ary heap, showed great promise as it reduced the number of potential cache misses by cutting the height of the tree in half.

The 4-ary Heap was then implemented and optimized for greater cache performance along with the Binary Heap.

The optimized, as well as the non-optimized 4-ary Heap, performed better than their binary counterparts in our tests on large inputs. The optimized 4-ary Heap was almost 30% faster than the original Binary Heap and 10% faster than the non-optimized 4-ary Heap. The non-optimized 4-ary Heap was 17% faster than the original Binary Heap. The optimization did not work for the Binary Heap.

While the cache-optimization did improve the performance of the 4-ary Heap it did not work as intended. It was suggested that this difference in performance was instead due to a difference in response to compiler optimizations, specifically function cloning.

It was concluded that while the optimized 4-ary Heap was the fastest in our tests, the results may vary from machine to machine, as such it might not be a wise choice for a software that is intended to be used on a variety of machines. Instead, it would be wiser to replace the Binary Heap with a non-optimized 4-ary Heap.
## Contents

1 Introduction .......................................................... 6  
   1.1 The Principles of Locality ....................................... 8  
       1.1.1 Spatial Locality ........................................... 8  
       1.1.2 Temporal Locality ........................................ 8  
   1.2 Methodology .................................................... 9  

2 Heaps and priority queues ........................................... 10  
   2.1 The URDME priority queue ....................................... 11  
   2.2 Implicit Heaps .................................................. 11  
   2.3 Binary Heap .................................................... 12  
   2.4 d-ary Heaps .................................................... 13  
   2.5 Binomial Trees ................................................ 16  
   2.6 Binomial Heap ................................................ 16  
   2.7 Fibonacci Heap ............................................... 18  
   2.8 Heap Summary ................................................ 19  

3 Optimizing implicit heaps ......................................... 20  
   3.1 Cache Alignment ............................................... 20  

4 Comparisons .................................................................. 23  

5 Conclusions and Future Work ....................................... 31
1 Introduction

A priority queue is an abstract data type that can be viewed as a queue of elements where each element is associated with a priority[3]. The priority then determines the placement of each element in the queue, with the most extreme priority at the front of the queue. This can be either the lowest priority or the highest priority, depending on how you view priority.

As the priority queue is an abstract data type it does not correspond to any specific implementation, unlike a data structure which is the specific implementation of some abstract data type. An abstract data type such as the priority queue instead describes some functionality from a users’ point of view [4]. In the case of abstract data types, the user might be a programmer who uses a priority queue from some software library.

The functionality defined by a priority queue are is follows:

1. *Is-empty* - Check if the queue contains no elements
2. *Insert* - Insert an element into the queue with a set priority
3. *Pop-min* - Remove and return the highest priority element from the queue

An additional operation, *Peek*, which returns the highest priority element from the priority queue without removing it, is sometimes said to be a part of the priority queue as well. Data structures that implement the priority queue often support other operations as seen in section 2.

The scientific software URDME[8] implements a different priority queue. It is what could be called a min-queue, where the element with the lowest priority is the element at the front of the queue. This priority queue only has three public operations.

These operations are:

1. *find-min* - Find the element with the minimum priority
2. *update-priority* - Decrease or increase the priority value of an element
3. *Initialize* - Initialize the priority queue

In this thesis, we will attempt to find a better priority queue implementation for URDME. This will be done by comparing and evaluating different implementations and optimizations. We will also consider the cache performance
of the priority queues, or as Peter Norvig put it, we will remember that there is a "Computer" in Computer Science[1]. As the difference in execution time between a cache miss and hit can be anywhere from twofold to tenfold or more[2], rather than viewing priority queues as just the sum of their operations, we will reason about and try and predict cache performance. This will be done by considering the priority queue implementations in terms of the *Principles of Locality*. Some questions that will be addressed during this process are:

1. Can we find a better priority queue for URDME?
2. Can we improve the cache behavior in URDME’s priority queue?
1.1 The Principles of Locality

The principles of locality are based on observations regarding memory access patterns that computer engineers have made over the years[14, p. 62]. These observations can be summarized as following: memory access patterns are not random, accesses tend to repeat themselves in time, and accesses are often followed by accesses to nearby addresses in the memory address space[14, p. 63]. The principles of locality are split up according to the two observations Spatial Locality and Temporal Locality. Computer architecture is designed with these observations in mind and attempts to use them to improve performance.

1.1.1 Spatial Locality

Spatial locality refers to the phenomenon that memory accesses are often followed by accesses to addresses nearby in memory space. This arises from the programmers and the compilers’ tendency to lump objects together [14, p. 63][2, p. 457]. A good practical example of this is arrays. Arrays are allocated as a contiguous block in memory, if we access data at index $i$, we are likely going to access the data at index $i + 1$ in the near future. An obvious way to optimize this so that we do not have to pull every single piece of data into the cache just as we need it is to use some sort of lookahead. Processors implement this via something called cache lines. Cache lines are atomic storage units in the cache which are larger than most primitive data types, in modern processors cache lines are typically 64 Bytes. When data is accessed and loaded into the cache, nearby data is also loaded into the cache to fill up the entire cache line.

1.1.2 Temporal Locality

Temporal locality refers to the phenomenon that memory accesses tend to repeat themselves in time. It is strongly intertwined with spatial locality in the sense that prediction here is not only that nearby data will be accessed in the future but the same piece of data. Support for this is implemented by the very existence of caches, or the fact that data is loaded from main memory into the cache. Cache replacement policies such as LRU also adhere to this by evicting the least recently used cache line in the cache of a write miss.
1.2 Methodology

This thesis is a study of the performance of algorithms operating on data structures. Most studies that compare and analyze the performance of algorithms use the RAM model. In the RAM model all simple operations such as the arithmetic operations are considered to take one time unit, memory access is also considered to take one time-unit\cite{18, p. 31-32}. This does not at all reflect the memory hierarchies that are present in modern computers where the cost of memory access that results in a cache miss can be as much as twenty times more expensive than one that results in a cache hit\cite{2, p. 455}. In this thesis, we will explore data structures that have procedures that display identical worst-case performance when using the RAM model for analysis but have different execution times in practice. We will reason about the locality that they exhibit and with the help of tools such as Valgrind’s cache profiling tool CacheGrind and the Linux kernel profiling tool Perf we will attempt to prove these differences empirically. We will also simulate different caches using CacheGrind to see how this affects the performance.
2 Heaps and priority queues

In a priority queue, we seek to find the largest or smallest attribute of an element in a set that is frequently changing, due to insertions and deletions. In this text, this value of interest will be referred to as either a *key* or a *priority* interchangeably. There are several ways to go about this problem. A naive approach would be to simply store all elements in an array, do a linear search for $O(n)$ time, find the smallest element and store it as a variable. When the smallest element is updated repeat the linear search. A slightly less naive approach might be to first sort the list using merge sort for $O(n \log(n))$ time, updating the priority of an element such that the sorting holds will take $O(\log(n))$ time using binary search, but finding the element with the most extreme priority will always be done in $O(1)$ time.

This however seems rather redundant and costly, if we only care about the object with the smallest key in our set, does the entire set need to be sorted at all times? Heaps allow us to tackle this problem more efficiently by partly sorting the set of objects.

A heap is a tree-based implementation of a priority queue, as a heap is a tree-based data structure we will refer to its elements as nodes. Being a priority queue implementation heaps support all the functions specified by the priority queue. Heaps typically also support additional operations to those specified by the priority queue.

These basic operations are:

1. *find-min* - Find the node with the minimum value
2. *decrease-key* - Decrease the key value of a node
3. *merge* - Combine two heaps

Beyond supporting these operations a heap satisfies a constraint called the *heap property*, which simply states that every node in the tree has a key that is more extreme than or equal to the key of its parent node. This ensures that the root node is the node with the most extreme key[16].
2.1 The URDME priority queue

The priority queue that we seek to implement is slightly different, since it does not need to implement most of the aforementioned functions. The URDME priority queue does however support an additional operation that was briefly mentioned in the introduction and alluded to in the title of this thesis, update-priority, which decreases or increases the priority of an element at a given index.

The priority queue currently in use by URDME is a Binary Heap implemented in C, which will be discussed in more detail in section 2.3. According to the RAM model the worst-case costs of the Binary Heap currently implemented by URDME are as follows:

1. find-min - $\mathcal{O}(1)$
2. update-priority - $\mathcal{O}(\log(n))$
3. Initialize - $\mathcal{O}(n)$

Most priority queue alternatives that we will be looking at has a similar worst-case cost for these operations using the RAM model. As mentioned in section 1.2 this is not something we will concern ourselves with to any great extent. Instead, the focus will lie on the locality that these structures exhibit.

2.2 Implicit Heaps

Implicit Heaps are heaps that are not explicitly defined by some data structure but implicitly defined in an array. This drastically reduces the overhead of heaps as most heaps allow for nodes to have at least two children, which means that the node object would need to contain at least two pointers. Because implicit heaps are defined in arrays, this means that they are allocated as a contiguous block of memory which is great for the spatial locality. Explicit heaps on the other hand exhibit poor spatial locality as each node will be allocated separately. There are more arguments to be made when considering explicit heaps and caches, the higher constant factor involved makes it so that the explicit heaps will fill the cache quicker which results in more cache misses for equally sized heaps.
2.3 Binary Heap

A Binary Heap is a type of heap that takes on the form of a binary tree. It must satisfy the heap property which we discussed at the end of section 2, and on top of that, the Binary Heap must satisfy an additional constraint, the shape property. The shape property implies that the Binary Heap must be a complete binary tree, that is, all levels of the heap must be filled, except for the last level, which is filled from left to right.

Figure 2.1: A Binary Heap (max-heap) as a tree

Figure 2.2: The same heap as figure 1 but implicitly defined in a contiguous array

The Binary Heap has some nice properties due to the shape property. Every layer of the heap, except the root, has a layer with twice as many nodes above it. From this we can conclude that a node at index \( i \), will have a parent with index \( \left\lfloor \frac{i-1}{2} \right\rfloor \). Vice versa is true for the children of a node, they can be found at index \( i \cdot 2 + c \) where \( c \in \{1, 2\} \). This relationship between parent and child nodes allows us to implicitly define the heap. Instead of explicitly
defining the tree structure we can store all our nodes in a contiguous array, in the order which we would encounter them in a left to right traversal of the tree. This significantly reduces the memory overhead of our data structure and improves the locality as we mentioned in section 2.2. For a heap of size \( n \) the explicit definition seen in figure 2.1 will use \( 3n \) more pointers since every node would have a pointer to its children and parent.

### 2.4 d-ary Heaps

d-ary heaps are a generalization of Binary Heaps in which can have \( d \) children instead of 2 as with the Binary Heap. A Binary Heap can also be viewed a special case of d-ary Heap, the case when \( d = 2 \). An upside to d-ary heaps with \( d > 2 \) compared to Binary Heaps is that upwards percolation is faster as it takes \( \log_d(n) \), the height of the tree, element compares for any d-ary heap, including the Binary Heap[15]. d-ary Heaps may also have better cache behavior, depending on the choice of \( d \), which can allow them to run faster in practice than other heaps which have better theoretical performance using the RAM model [17]. A popular choice for \( d \) is 4 because this is said to improve the cache performance of the heap. In figure 2.3 we see explicit d-ary heap where \( d = 4 \), and in figure 2.4 we see the same heap but implicitly defined.

Figure 2.3: A 4-ary Heap(min-heap) as a tree

The implementation of a d-ary Heap is identical to that of a Binary Heap, it is just a generalization. A node at index \( i \) will have a parent with index \( \lfloor \frac{i-1}{d} \rfloor \) and potential children will be at index \( i \cdot d + c \) where \( c \in \{1, 2, .., d\} \).
Figure 2.4: The same heap as in figure 2.3 but implicitly defined

Just like with the Binary Heap d-ary Heaps can be implicitly defined in a contiguous array, in the order that we would encounter the nodes during a left to right traversal of the tree.

Why is $d = 4$ said to improve cache performance compared to the Binary Heap? To get an intuition as to why we must take a look at what implicit 4-ary and Binary Heap might look like in the cache. When an update increases the priority of a node in our heap, that node must be percolated down. When percolating down we must make sure that we do not violate the heap property, to ensure this a node that is moving down a level must always swap places with the smallest of its children. To find the smallest child of a node the percolate down operation must always compare all the children of a node. In figures 2.5 and 2.6, we can see how a binary and 4-ary Heap might be split among cache lines.

For every step of percolate down in the Binary Heap, at least one cache line needs to be fetched and in $\frac{1}{4}$ of the cases two cache lines need to be fetched. Some back-of-the-envelope calculation tells us that in $\frac{3}{4}$ cases exactly one cache line is fetched, and in $\frac{1}{4}$ cases two cache lines are fetched giving us a formula for the average sibling pair that looks like $\frac{3}{4} + \frac{2}{4} = \frac{5}{4}$. 
Figure 2.6: An implicit 4-ary Heap split between cache lines

Attempting the same back-of-the-envelope calculation for the 4-ary Heap we end up with $\frac{1}{2} + \frac{2}{2} = \frac{3}{2}$. What is going on here? From the looks of it the Binary Heap actually has fewer cache line fetches and therefore should have fewer misses since $\frac{3}{2} > \frac{5}{4}$. But this represents the average number of cache line fetches per set of siblings, a Binary Heap has twice as many sibling pairs as a 4-ary Heap has sibling quadruples in a heap of the same size. So for a heap of size $n$ the estimated amount of cache line fetches while percolating down the entire heap would be $\log_4(n) \cdot \frac{3}{2}$ in the case of the 4-ary Heap and $\log_2(n) \cdot \frac{5}{4}$ in the case of the Binary Heap. Dividing both sides by $\log_4(n)$ we get the relationship $\frac{3}{2} < 2 \cdot \frac{5}{4}$ meaning that there is roughly $\frac{5}{3}$ times as many cache line fetches in the Binary Heap than in the 4-ary Heap when processing an equal amount of data while percolating a node down.
2.5 Binomial Trees

In order to understand Binomial Trees one must first understand Binomial Trees. A Binomial Tree of degree 0 is just a single node. A Binomial Tree of degree 1 is formed by connecting the roots of two Binomial Trees of degree 0. This process is recursive, a Binomial Tree of degree \( k \) is formed by joining two Binomial Trees of degree \( k - 1 \). In figure 2.7 we can see a collection of binomial trees of degrees 2, 1, and 0.

![Figure 2.7: A collection of binomial trees of degrees 2, 1, and 0](image)

2.6 Binomial Heap

A Binomial Heap is a type of heap that unlike the Binary Heap, which takes on the form of a Binary Tree, takes on the form of a collection of Binomial Trees. But much like the Binary Heap, the Binomial Heap must also satisfy the heap property constraint. In a binomial heap, there is at most one Binomial Tree of a given degree. Because there can only be one Binomial Tree of a given degree in a Binomial Heap, and the number of nodes in a Binomial Tree of degree \( k \) is \( 2^k \), the binary representation of the number of nodes in the heap will tell us which degree of Binomial Trees are present in the heap. For example, a Binomial Heap with 9 nodes gives us \( 9 = 1001 \) indicating that we have one tree of degree 3 and one of degree 0 in our collection. The binary representation of the number of nodes in the Binomial Heap gives great insight into the structure of the heap. This can also be used to understand the operations on a Binomial Heap. Inserting into a Binomial Tree works just
like the addition of binary numbers. Inserting a new node into the Binomial Heap with 9 nodes equates to $1001 + 0001 = 1010 = 10$, which is the number of nodes after insertion. Every time that we have to carry the 1 a merge of two Binomial Trees is preformed, when performing this merge its important to make sure that the heap property is not violated.

Binomial Heaps are typically explicitly defined, but there are ways to implicitly define them. However such implicit definitions do not reduce the overhead as drastically as it does with the Binary Heap if we care about maintaining the worst-case costs. One such implementation where the Binomial Heap is implicitly defined as a contiguous array practically requires $lg^*n$ extra pointers and some extra information to be stored, compared to the implicit definition of the Binary Heap [5].
2.7 Fibonacci Heap

A Fibonacci Heap is a forest of rooted trees, each fulfilling the *heap property* constraint. We will always keep track of which of the trees in our forest has the most extreme value, min or max depending on which we are interested in. Theoretically, the Fibonacci Heap is really fast compared to the other heaps, especially if the extract-min operations are few compared to the other operations. However, practically the large overhead of the Fibonacci Heap makes it less desirable for most applications [6, p. 507]

Unlike the typical implementation of the Binomial Heap, which is also based on a forest of trees existing in a root list, we do not care about having only one tree of each degree at all times. This allows for inserting any item as a tree of degree 0 which can be done in $O(1)$ time. Instead, we only care about this after we perform the pop-min operation, after performing pop-min, we merge trees, allowing at most one of a given degree. A similar approach is taken to updates which we are particularly interested in. If the key of a node is updated such that it violates the heap property that node is removed from the sub-heap and moved to the root list, its parent is marked a loser. If a parent node that is marked a loser loses yet another node, then the parent node is also moved to the root list. This is done to avoid a situation where we have a shallow tree with a lot of nodes since extracting the minimum value would then be expensive. This behavior is often referred to as lazy and is based on the simple idea that doing work in bulk is often more efficient.
2.8 Heap Summary

The binomial heap has been discarded as it does not really improve anything. The risk of having to do $O(\log(n))$ comparisons when updating might be slightly lower, but due to the binomial heap being explicitly defined the spatial locality is a lot worse then what we see in an implicitly defined heap. The Fibonacci heap is an honorable mention because of its theoretically impeccable performance. The lazy behavior when it comes to updates is something that we could potentially benefit from. But as many sources discredit the Fibonacci heaps practical performance[6, p. 507][10], unless the number of find-min operations are few in comparison to the number of update operations[6, p. 506]. It was decided to also rule out the Fibonacci heap as we can not say anything certain about the ratio between update and find-min operations. Binary Heaps or d-ary Heaps are often the most desirable heaps for practical applications. Due to their low constant factor and relatively good worst-case performance. The ability to implicitly define heaps show additional upside, as they exhibit a higher degree of Spatial Locality. With our cache-centric approach to priority queues, it does make sense to go for implicitly defined structures. We choose to further explore how one can optimize these types of heaps because if any optimization is possible we will have improved on what we currently have and end up with a faster priority queue.
3 Optimizing implicit heaps

There are a few obvious ways of optimizing implicit heaps. Since the parents and children in the node are defined by implicit relationships using integer division and multiplication respectively we can opt for using bit-shifting instead which should be faster at least for division [11, p. 139].

A different optimization is similar to the lazy behavior of the Fibonacci heap described in section 2.6. Instead of initializing the heap by repeated insertions in $O(n \log(n))$ time we heapify an unsorted array by percolating down all child nodes in a bottom up fashion in $O(n)$ time. Although one can hardly call this an optimization as this was proposed back in 1964 by Floyd [9], the same year as the Binary Heap was first introduced and has since been standard practice. Interestingly enough this optimization was later disputed by LaMarca and Ladner [17] who suggested that repeated insertions could be a better option due to better cache behavior. They also empirically proved this to be the case for very large heaps.

Another optimization would be to try to improve the cache behavior of the structures as a whole, not just certain operations. This is what we will discuss in section 3.1.

3.1 Cache Alignment

When we viewed sibling quadruples and pair together with cache lines while comparing the 4-ary Heap and the Binary Heap in section 2.4 we took interest in the average number of cache line fetches per sibling pair or quadruple. This was interesting because more cache line fetches means more potential cache-misses which is costly. In this section, we will explore a way to reduce the number of cache line fetches per sibling set and get rid of some assumptions that were made previously.

When we saw cache lines in section 2.4 there was an assumption made as to where the cache line begins, we assumed that the cache line started at the base address of the array. We know that a cache line is a contiguous block of memory that is typically 64 bytes in modern processors, but what 64 bytes are selected? To understand this we first must understand how addressing in caches work. In modern set-associative caches, the address of some data is split up into 3 sections, tag, set, and offset. The tag bits are used to uniquely identify a cache line in a block, the set bits are used to identify what set that data belongs to and the offset bits are used to address.
bytes in the cache line. As data in the cache is byte-addressable the index bits of the offset portion must be able to address every bit in the cache line, therefore we need \( \log_2(\text{cache line size}) \) bits in the block-offset. The amount of set bits required is determined by the size of the cache and its associativity, for example, if the cache is 65536 Bytes (64kB), the line size is 64B and the cache is 8-way set associative. Then we need \( \log_2(64) = 6 \) bits for the offset, the number of bits required for the set portion of the address is given by \( \log_2(\text{number of sets}) \), to calculate the number of sets we first need to calculated the number of cache lines, \( \text{number of lines} := \frac{65536}{64} = 1024 \), \( \text{number of sets} := \frac{\text{number of lines}}{\text{ways}} \iff \frac{1024}{8} = 128 \), so we need \( \log_2(128) = 7 \) bits for the set portion of the address. The rest of the address \( 64 - 6 - 7 = 51 \) bits can be used for the tag.

<table>
<thead>
<tr>
<th>Tag 51 bits</th>
<th>Set 7 bits</th>
<th>Offset 6 bits</th>
</tr>
</thead>
</table>

Figure 3.1: A example of how a address is partitioned for cache indexing

This implies that cache lines are N-byte aligned where N is the byte size of a cache line and is crucial for cache alignment.

As arrays are allocated as a contiguous block of memory if we can control the alignment of the base address of the array we can make sure that our array is allocated at the beginning of a cache line. In the C programming language, it is possible to control what base address an array is allocated at using the `aligned_alloc` function [13, p. 347]. It is worth noting that aligning the array like so is not a cache-oblivious optimization as it assumes 64B cache lines.

A good way to reduce the number of cache line fetches would be to make sure that siblings are always allocated on the same cache line, this is something that was proposed by LaMarca and Ladner [17]. The way LaMarca and Ladner proposed to do this was by padding the head of the array with some sentinel values. So if the array consists of 8-byte elements and is allocated at a 64-byte aligned address. Then the root node of the implicit heap will be on a cache line of its own, but all siblings will be on the same cache line.
This way the number of cache line fetches required to percolate a node down the entire heap directly corresponds to the height of the heap, so $\log_4(n)$ in the case of the 4-ary Heap and $\log_2(n)$ in the case of the Binary Heap, where $n$ is the size of the heap in the number of nodes.

Figure 3.2: An example of a cache aligned implicit 4-ary Heap with seven sentinel values and 8-Byte data

Another possible padding for cache alignment only uses three sentinel values, with that padding the root is on the same cache line as its children.

Cache aligning the Binary Heap is possible with only one sentinel value. This has an additional upside as it requires fewer operations to index the children and parents of nodes. Instead of having to use $\lfloor \frac{i-1}{2} \rfloor$ to get the parent-index of the node with index $i$, we can now use $\lfloor \frac{i}{2} \rfloor$, accessing the left and right child of a node at index $i$ is done with $2i$ and $2i + 1$ respectively, instead of $2i+1$ and $2i+2$. Getting rid of one subtract and one add operation might seem insignificant, but mind that these are operations that are used very frequently. However modern CPUs have more complex adders that can perform addition faster, so the difference might be negligible [7, p. A-37].

Figure 3.3: A example of a cache aligned implicit Binary Heap with one sentinel value and 8-Byte data
4 Comparisons

The evaluation is focused on the update function as the other function of interest \textit{find-min}, is always a constant time operation. All tests are run using the -O3 optimization flag on an Intel i5 3230M processor with a 64kB L1 cache, 256kB L2 cache (non-inclusive), and a 3MB inclusive L3 cache, unless otherwise specified. Heaps of sizes $N \in \{100, 1000, 10000, 50000, 100000, 150000\}$ will be used in the evaluation. The heaps are tested by 50 000 iterations of assigning a new random value to the root node and updating, as well as 50 000 iterations of assigning a new random value to the right-most leaf node and updating the heap.

In the mean execution time plot of Figure 4.1 each heap was evaluated 10 times and from this, a mean execution was extracted. As we can see the aligned 4-ary Heap had the best execution time. In Figure 4.1 we can also see the cache behavior from the same execution. Looking at these two graphs it seems as if the cache-optimization did not affect anything. But the execution time in figure 4.1 tells a different story, at least in the case of the 4-ary Heap. The L1 cache behavior was identical between the optimized and non-optimized for both the binary and 4-ary Heap. In the case of the LL cache behavior the optimized and non-optimized 4-ary Heap exhibit the same behavior, but in the case of the Binary Heap, the aligned version seems to have way better behavior for most heap sizes, despite having the slowest execution time. What dictates the performance here is not the cache-misses, which is not surprising as the heaps fit in the cache. What is surprising however is the difference in cache behavior between the optimized Binary Heap and non-optimized.
Figure 4.1: Execution time and cache statistics of the various heaps
What seems to dictate performance here is the number of instructions as seen in figure 4.1. A possible explanation for this in the case of the 4-ary Heaps is that when iterating over a sibling quadruple there is a 50% risk that we have to fetch two cache lines instead of one. As seen in figure 4.2 every other sibling quadruple suffers from a split among two different cache lines. This would mean that the 4-ary Heap has to fetch more cache lines as related data is poorly structured in memory.

When using both Valgrind’s tool Cachegrind and Perf for analysis we saw no difference in the number of cache-references. When using Cachegrind to annotate the code to see where the difference in the number of instructions arises from we did however find an interesting snippet:
This snippet is from a test that tests if the heap property constraint holds, the test was used to assert this in every iteration of our evaluation of the update function mentioned at the start of this section. Comparing lines 7 and 17 of the snippet we can see that a large portion of the difference in instructions arises from accessing the right-most sibling in the quadruple. While there are some unexplained differences this annotated snippet seems to support our theory that the increased number of instructions arises from the fact that every other sibling quadruple is split among cache lines.
To get the maximal effect from cache aligning heaps one should have a heap that is large enough to fill the entire cache. As having to access main memory is what is truly costly and not just extra cache line fetches. Modern caches are quite large, typically 3MB or larger, this requires a really big heap size. To accommodate this we decided to simulate a smaller cache instead of working with humongous heaps. In the figures below we have simulated a cache with 64kB L1 cache, 256kB L2 cache, and 64kB LL(L3) cache. If the extra instructions do indeed arise from the extra cache line fetches, then we should be able to see a difference in cache-misses using the simulation. These simulations were run on three heap sizes 50 000, 100 000, and 150 000, using the 4-ary and aligned 4-ary Heap.
Figure 4.3: Number of instructions and cache-misses versus heap size in elements simulating a 64kB LL-cache
As seen in Figure 4.3 even though these heaps do not fit in the cache there is no significant difference in the number of cache-misses. Which is in contrast to what LaMarca and Ladner showed in their paper [17]. This does however explain why the aligned binary performed worse than its unaligned counterpart, the cache alignment did not work the way that it was intended. A similar result was reached by Hendriks who attributed this to cache management logic in modern computers being more complex and better at predicting what pieces of memory will be needed[12].

So if the cache alignment did no work, why was the aligned 4-ary Heap faster then its aligned counterpart? Looking yet again at the number of instructions for the heaps running with a simulated 64kB LL-cache in figure 4.3, it seems as if it is still the number of instructions that dictate the performance.

If the reduced number of instructions is not a direct effect of the cache alignment, what is the cause of this? Some experimentation with compiler flags revealed that there is a difference in response to different compiler flags between the optimized and non-optimized 4-ary Heaps. The greatest difference response was caused by the optimization flag \texttt{-fipa-cp-clone}. This optimization flag seems to be the cause of the difference in the number of instructions between the optimized and non-optimized 4-ary Heaps, visualized in Figure 4.4
Figure 4.4: The number of instructions in the respective heaps when running the tests on a 50 000 element heap with/without the -fipa-cp-clone flag
5 Conclusions and Future Work

As seen in figure 4.1 both the 4-ary and aligned 4-ary Heap outperforms the Binary Heap currently implemented by URDME in terms of execution time. The reason that the aligned 4-ary Heap outperforms the regular 4-ary Heap seems to have to do with compiler optimization and function cloning. Since the optimization does not work as intended and we have yet to provide a clear reason as to why it works at all it is regarded as unsafe for implementation. It has only been shown to work on one machine with one compiler version (GCC 6.3) and is therefore not suitable for implementation in a program that is supposed to run on a wide variety of machines with different configurations. The regular 4-ary Heap is considered safe for implementation as it consistently performed well with on different configurations. It is also a good 17% faster than the original Binary Heap implemented by URDME. This thesis has been a testament to the risks of cache optimizations, what is an optimization on the hardware of today might have an adverse effect on the hardware of tomorrow. Even if the cache alignment would have worked it might not have been worthwhile to implement in URDME as its performance could decay over time when changes are made to compiler optimization techniques and computer hardware. For the same reason, empirical performance evaluations of data structures do not age well and should be viewed with great skepticism.

There is some future work to be done as the approach we took to caches in this paper, which excluded explicitly defined heaps, appears to be outdated. This opens up possibilities for finding a better priority queue. There is also a lot of optimizations done to heaps that were not considered in this thesis, such as concurrent heaps. A detailed explanation as to why the cache-optimizations did not work, and why the Binary Heap and 4-ary Heap responded differently to it, is also yet to be given.


References


