Cache-Oblivious Algorithms in Practice

Master's Thesis by:

Jesper Holm Olsen dunkel@dunkel.dk Søren Christian Skov srenskov@mail.dk

Department of Computer Science University of Copenhagen

December 2nd, 2002

Abstract

The memory of contemporary computers is structured in a hierarchy of successively larger, slower, and cheaper memory levels. Each level contains a working copy—or *cache*—of the level above. Recent developments in processor and memory technology imply an increasing penalty if programs do not take optimal advantage of the memory hierarchy. To alleviate this, the notion of *cache-oblivious* algorithms has been developed. The main idea behind cache-oblivious algorithms is to achieve optimal use of caches on all levels of a memory hierarchy without knowledge of their size. The purpose of this thesis is to examine cache-oblivious algorithms from a practical point of view. The thesis consists of a discussion of the theory, and a thorough benchmarking of some of the most recently published cache-oblivious algorithms, in the form of two priority-queue algorithms.

The cache-oblivious theory has, so far, not incorporated the virtual memory system. We show that each of the levels in the virtual memory system can be seen as a separate level of cache, and is therefore also encompassed by the theoretical model.

Several practical details that are not explained in the original articles, are described in order to implement the selected algorithms. Most importantly, we clarify how the algorithms should be constructed to use linear space.

We furthermore develop a new optimal cache-oblivious algorithm for a priority deque, based on one of the cache-oblivious priority queues.

Our results show, that for the cache-oblivious algorithms used in our casestudy, the extra work incurred by making algorithms cache oblivious is too big, for them to be competitive on all levels in the memory hierarchy.

Cache-oblivious algorithms do outperform traditional RAM-model algorithms when working on data sets larger than main memory, but in the lower levels of the memory hierarchy, the traditional RAM-based algorithms are faster, due to a smaller amount of work. The cache-aware implementations exhibit good use of the caches without too much extra work, and are therefore significantly better.

Based on our investigation of two priority-queue algorithms, we conclude that the cache-oblivious approach is currently not able to offer a competitive priority queue, except when working on data sets larger than main memory.

Contents

1	Intr	oduction	n	1
	1.1	The Ev	volution of Computers	2
	1.2	An Inti	roduction to Memory Hierarchies	2
	1.3	The Vi	rtual Memory System	5
	1.4	Memor	ry Performance	7
		1.4.1	Measuring Memory Latency	8
		1.4.2	Development in Memory Latency	8
	1.5	Previou	us Work	10
	1.6	How to	Read this Thesis	11
2	Ana	lyzing L	O Complexity	12
	2.1	Extern	al-Memory Algorithms	13
	2.2	Cache-	Oblivious Algorithms	15
	2.3	The Ide	eal-Cache Model	17
		2.3.1		18
		2.3.2	Optimal Replacement	19
		2.3.3	Automatic Replacement	21
		2.3.4	Associativity	22
		2.3.5	Multiple Levels of Memory	22
		2.3.6	The Virtual-Memory System	23
	2.4	An Inti	roduction to Cache-Oblivious Techniques	25
3	Cac	he-Obliv	vious Priority Queues	28
	3.1	The Di	istribution Heap	29
		3.1.1	Push	32
		3.1.2	Pull	34
		3.1.3	Insert and Extract	35
		3.1.4	Space Complexity	38
		3.1.5	Delete	39
		3.1.6	Global Rebuilding	40
		3.1.7	The Constant c	41
		3.1.8	Elements with Identical Priorities	41
		3.1.9	Finding the Median Cache Obliviously	42

		3.1.10 STL Interface	ł
		3.1.11 Pseudo Code	5
	3.2	The Funnel Heap 48	3
		3.2.1 Complexity of Insert and Extract	3
		3.2.2 Dynamic Structure	ł
		3.2.3 Space Complexity	5
	3.3	Construction of Priority Queues	5
		3.3.1 Distribution Heap	5
		3.3.2 Funnel Heap 57	1
	3.4	Sort Method	3
	3.5	Limited Address Space)
4	A C	ache-Oblivious Priority Deque 61	L
	4.1	RAM-model Priority Deques	2
	4.2	A Cache-Oblivious Priority Deque	ŀ
		4.2.1 Push	5
		4.2.2 Pull)
		4.2.3 Insert, Extract_min and Extract_max	
		4.2.4 Construction)
	4.3	Summary	3
5	Mea	suring Performance 74	ł
	5.1	Previous Work	ł
	5.2	Choosing Benchmark Tools	5
		5.2.1 Timing	5
		5.2.2 PAPI	5
		5.2.3 Page Faults	7
		5.2.4 Profiling with gprof	3
	5.3	Methodology 78	Ś
	0.0	5 3 1 Choosing Benchmark Data 78	Ś
		5.3.2 Measurements 78	Ś
		5.3.2 Choosing Hardware Platforms 70	ý
		5.3.4 Configuring the Benchmark Computers	í
		5.3.5 Validity 81	
		5.5.5 Valuty	-
6	Prep	paring Benchmarks of Priority Queues 83	5
	6.1	Sequences of Data	3
	6.2	Data Sizes	5
	6.3	Types of Elements 85	5
	6.4	Choice of Compiler	5
	6.5	Choosing Competitors	5
		6.5.1 RAM-model Priority Queues	7
		6.5.2 Cache-Aware Priority Queues	7
	6.6	Practical Issues)

7	Performance Investigation	92
	7.1 Sorting Method	93
	7.2 Locality of Data	96
	7.3 Rebuilding and Sweeping	103
	7.4 TLB and the Virtual Memory Hierarchy	104
	7.5 Using 64-bit Computers	107
	7.6 Summary	108
8	Conclusion	111
	8.1 Further Work	114
	Bibliography	115
	List of Figures	120
	List of Algorithms	122
	List of Symbols and Abbreviations	123
A	Source Code	124
B	Benchmark Results	126

Preface

This paper is a Master's Thesis written by Jesper Holm Olsen and Søren Christian Skov at the Department of Computer Science, University of Copenhagen. The thesis is written with the assumption that the reader has an understanding of computer science corresponding to a graduate level.

The supervisor for this thesis was Jyrki Katajainen, who we thank for incredibly detailed feedback and many inspiring and constructive discussions on the thesis itself, as well as many other subjects. We also thank Lars Arge and Bryan E. Holland-Minkley for clarifying details of space consumption in their data structure [6] as well as Gerth Stølting Brodal for the same issue in [14]. For helping us with proofreading we thank Martin Koch, Jesper Rasmussen, Maz Spork, and Siri Willassen. Finally, we thank both the IT-staff at DIKU for lending us a Sun Ultra-SparcII computer and Per Leslie Nielsen for lending us an AMD Athlon computer for use in benchmarking.

In order to avoid handing in several binders of paper we have chosen to include the source code and benchmark results on a supplemental CD-ROM. The content of this can also be downloaded from:

www.dunkel.dk/thesis

Jesper Holm Olsen & Søren Christian Skov Copenhagen, December 2002

Chapter 1

Introduction

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1 1/2 tons.

- unknown, Popular Mechanics, March 1949

The main purpose of this thesis is to examine the practical relevance of an emerging theory of algorithmic design called *cache-oblivious algorithms*¹. As the notion of cache obliviousness is quite new—just a few years old—there only exists few practical studies of how the theory works in practice. The majority of the available literature is focused on theoretical issues and very little work have been done on examining the relationship between theory and practice. A brief overview of the few articles which do incorporate benchmark results is found in Section 1.5 on page 10.

Because of the relatively sparse amount of real-life practical benchmarks of cache-oblivious algorithms, we feel that this is an interesting and important field for a more comprehensive treatment. An important part of this thesis is therefore to conduct a thorough benchmarking of the algorithms we choose to examine. For this purpose we choose two of the most recently published cache-oblivious algorithms, in the form of two priority-queue data structures. Priority queues are commonly used data structures useful in many applications and an effective cache-oblivious priority queue is therefore an interesting subject of research.

We start this chapter with a brief overview of the evolution of computers with focus on memory technology. Following this, we give a brief overview of previous work, and finally present a guide for reading this thesis.

¹The cache-oblivious theory was introduced by Prokop in 1999 [44], and later published and refined in an extended abstract by Frigo et al. [22].

1.1 The Evolution of Computers

The first computers only had a flat memory structure—that is, there were only the main memory in which the CPU could load and store words. This was acceptable because the bottleneck was the relatively slow CPUs.

The way theoretical reasoning regarding the performance of programs was developed, reflected this reality as it only took the number of instructions into account—memory references were seen as having a constant cost. The *complexity* of programs was measured in the so-called RAM model (Random Access Machine) [3], consisting of one CPU and a unit-cost random-access memory, which made analysis simpler and was still a reasonable abstraction for real-life hardware.

Since then, the development of the speed of CPUs has pretty much followed the pattern described by Moore's Law, which states that integrated circuits will contain up to 55% more transistors every year [39]. More transistors means that more features can be incorporated in the design, which in practice means a great improvement in speed. In contrast, the development of memory speed has been extremely slow with an improvement of approximately 7% per year, as described by Hennessey & Patterson [25].

This development implies that memory in computers will become much slower compared to CPU, over time. A plot of these projections in [25, figure 5.2] shows that CPUs today are expected to be approximately 8,000 times faster than in 1980, whereas the memory is only approximately 4–5 times faster. Current development in computer hardware indicate that this trend will continue, so new ways of thinking are needed.

1.2 An Introduction to Memory Hierarchies

This very obvious disparity has required changes in the way computer hardware is designed. The common way of handling the problem has been the introduction of a *memory hierarchy*. Here a small and fast, but expensive, type of memory is placed close to the CPU, and increasingly slower and cheaper types of memory are used in larger layers further away from the CPU. A small, fast level of memory acts as a *buffer* for a larger and slower level of memory. This kind of buffer is called a *cache*. A typical memory hierarchy is illustrated in Figure 1.1.

The number of levels in the memory hierarchy differs from the different architectures, but in the remainder of this thesis we assume that there are two levels of cache, unless we explicitly specify otherwise. This is done to simplify the descriptions and because a number of architectures actually have two levels. The cache levels are often abbreviated, such that level 1 caches are called L1, level two is called L2, and so on. Normally the term cache refers to the small, fast levels of memory close to the CPU, but today it is also common to refer to a layer in the hierarchy as the cache of the layer above—for example main memory can be seen as a cache for data stored on disk.



Figure 1.1: A typical memory hierarchy in contemporary computers.

There is a lot of technical detail involved in caches which have an impact on cache behavior and performance. Some will have a significant influence on the execution time of programs. On the other hand, there are also a lot of details which are not interesting from a programmer's point of view. Nevertheless, it is important to be aware of the behavior of caches, so we will try to cover the most relevant aspects of caches without going into too many technical details.

The construction of memory hierarchies is based on the observation of *the locality of reference*: The first part of this is, that once a program has referenced a memory location, it is likely to use this location again several times within a short time (*temporal* locality). The second part is that once a program has referenced a memory location, it is likely that it will access nearby memory locations shortly after (*spatial* locality).

The overall design of a cache is that it can contain a number of *cache blocks* (the part of a cache that can store a cache block is called a *cache line*). The blocks will generally hold more than one word. This is done to benefit from spatial locality. The performance improvement is achieved when a program access data that is already stored in the cache.

When a program requests data that is not in the nearest memory level, it is called a *cache miss* and the requested word has to be fetched from the next (and slower) memory level.

An interesting question is how long does a cache block stay in a cache. To achieve the best results the blocks stay in the cache as long as possible, so a block is first removed when another block needs to be stored in the same location.

The placement of a block in the cache is influenced by two things: The *replacement policy* and the *associativity* of the cache.

The associativity of a cache puts some restrictions on which blocks can be

chosen by the replacement policy. Ideally a block can be placed anywhere in a cache. These types of caches are said to be *fully associative*. However, the cost of producing fully-associative caches in hardware is so high that most caches have limited the number of places they need to look for a cache block. Such caches are called *set associative*, and consists of a number of sets. The address is used to uniquely identify in which set a cache block can be placed. A cache that has room for x elements from the same set is said to be x-way set associative.

If the cache is already filled with blocks, a block has to be removed from the cache every time a new block is fetched. The replacement policy determines which block is to be removed from the cache. Studies [50] have shown that the best replacement policy is where the *Least-Recently-Used* block is removed. Almost all caches therefore use the *LRU replacement policy*, or heuristics that achieve almost the same result in practice.

The disadvantage with a set-associative cache is that it might result in a higher number of cache misses than a fully-associative cache. This occurs if a program in a short time accesses more than x blocks which all belong to the same set, in an x-way set associative cache. This implies that the more elements of the same set a cache can contain, the better the performance.

A special case is a 1-way associativity also called a *direct-mapped* cache. As each cache block goes to a specific cache line, direct-mapped caches do not need a replacement policy. One way to achieve poor performance with a direct-mapped cache is by repeatedly reading two different cache blocks, that both map to the same cache line. Repeating this pattern N times would result in 2 cache misses in a set-associative cache, and in 2N cache misses in a direct-mapped cache.

When caching data stored on disk in main memory, the replacement policy is implemented in software, as a part of the operating system. In this case the extra resources needed to implement the software, such that the main memory is used as a fully-associative cache, is small compared to the cost of fetching a page from the disk. Therefore, most operating systems use the main memory as a fullyassociative cache. A cache miss in main memory is also called a *page fault*.

The type of cache misses that occur can be put into three different groups, depending on what caused it [25]:

- **Compulsory** cache misses are unavoidable. They occur when data is accessed the first time and there is therefore no way that it could have been stored in the cache.
- **Capacity** cache misses occur if the requested data has been in the cache, but was removed by the replacement policy. These type of cache misses can be reduced by improving the replacement policy, if a better exists, or by increasing the size of the cache.
- **Conflict** cache misses occur if the cache block for the requested data must be placed in an occupied cache line. This can occur in direct-mapped and set-

associative caches. By increasing the associativity, this type of miss is normally reduced.

Caches are *inclusive*—that is, data in one cache is also present in the larger levels in the memory hierarchy. This is desirable in order to exploit the memory hierarchy as effectively as possible: if a capacity or conflict miss occurs in level 1, the chance for a hit in level 2 is greater [25].

When talking about the speed of caches and the cost of accessing them, we will use the term *latency*. The latency is the time it takes to fetch the data, from when the data is requested by the CPU until the data arrives in the CPU, ready to be used. In a number of occasions we would like to compare this latency with the amount of work the CPU could have performed, while it was waiting. We measure this latency in the number of CPU clock cycles, and call this the *relative latency*.

The performance penalty imposed, if a program does not exploit the hierarchical memory design, forces the algorithm designer (and programmer) to develop his algorithms (programs) in a way that utilizes the cache hierarchy. This makes the original assumptions of locality a self fulfilling prophecy. Cache performance tuning is the study of how to make software live up to the expectation of temporal and spatial locality, so that the programs achieve the highest performance gain possible.

As the use of memory hierarchies has not been able to neutralize this big difference in development of computers, hardware producers use a range of other means to make memory usage more efficient.

One way is to make the CPUs super scalar. For a description of super scalar CPUs, see Hennessy et al. [25]. This approach attempts to enable the CPU to continue working, while data is fetched from memory. If it is possible to perform other instructions while the data is fetched, it is possible to "hide" some of the time used for fetching data. This is however only possible as long as there exists independent instructions that can be performed.

Other approaches involve reordering instructions, in order to execute instructions, which do not have to wait for the memory hierarchy, to implement branch prediction, and prefetching of data.

1.3 The Virtual Memory System

Another aspect of contemporary computers that influences the execution time of programs is the *virtual memory system*. This system releases the programmer from the burden of manual memory management, as well as providing protection between individual processes so that they cannot overwrite other processes' data. The idea is that each process is given its own *address space* which is only available for this particular process. The hardware and operating system then maintain a mapping from the *virtual address* to a *physical address*. This means that the process is able to use any part of its address space without interfering with other processes.

To support mapping from virtual to physical memory one or more *tables* are maintained by the operating system. To perform the translation, the virtual address is used as an index in the tables to locate the corresponding physical address. To minimize these tables both physical and virtual memory is divided into a number of *pages* typically with a size of 4 to 64 kilobytes. One virtual memory page is mapped directly to one physical page. This way the tables only have to contain one entry for each virtual page. The pages are similarly used as blocks in main memory, so that when any address in a page is accessed, the operating system ensures that the corresponding physical page is cached in the main memory.

Figure 1.2 illustrates how the virtual memory system works.



Figure 1.2: Illustration of a memory hierarchy with a virtual memory system.

When a process loads or stores data from an address, the virtual address is used to index the level 1 cache. While the cache is searched for a cache block matching this virtual address, the virtual address itself is converted to a physical address by the TLB. The level 1 cache now uses this physical address to validate whether a cache block found in the cache is actually the correct block, or a block from another process that just happened to use the same virtual address.

These tables of virtual-to-physical mapping are themselves stored in memory and is therefore normally found in the main memory, but they can also be swapped to disk because of resource limitations. In the worst case, a conversion from virtual to physical address therefore requires a disk access. However, if the address conversion always would require an access to main memory or disk, it would eliminate the benefits of having caches. This is avoided by adding a specialized cache that can contain entries from the virtual-to-physical conversion table. This cache is called a *translation look-aside buffer* or *TLB*. The TLB uses the property of locality of reference as any other cache. It ensures that an address conversion is cached such that subsequent conversions of addresses in the same page will result in a cache hit in the TLB. If a TLB miss occurs, the table in memory will be accessed, resulting in a main-memory access—or, if the page containing the table is swapped to disk, a disk access.

Due to the virtual memory system, there are a number of things that can happen on a memory access, which greatly influence the time it takes to fetch data. The combined time can be computed as:

average fetch time = L1 latency

- + frequency of L1 cache misses \times L2 latency
- + frequency of TLB misses × main memory latency
- + frequency of translation table misses in main memory \times disk latency
- + frequency of L2 cache misses × main memory latency
- + frequency of main memory cache misses × disk latency

The lookup in the level 1 cache and the TLB conversion of the address are done simultaneously. This is reflected by the fact that the cost of a TLB hit is not included in the computation, as it is hidden by the cost of the simultaneous level 1 cache lookup.

1.4 Memory Performance

The development of hardware makes it interesting to study the cost of accessing each level in the memory hierarchy. CPUs will be able to handle larger problems with their increased performance, but the memory technology cannot keep up to speed. Naturally, this lead to an increasing problems accessing the higher levels in the memory hierarchy, thus being dependent on slower memory. This disparity will therefore have two implications: Current algorithms that process *large* data sets, and are dependent on main memory and disks, will not gain the full speedup of the improved CPUs due to the increase in relative latency in caches. The other implication is that problems which today are CPU intensive will, as the CPUs get faster, become limited by the memory system. All in all, many algorithms are influenced by this development, and memory performance is becoming increasingly important.

We are therefore interested in looking at the actual development in the relative speed of CPU and memory in computers.

For practical reasons is is difficult to obtain the latencies in manuals or other documentation. This is because the relative latency of the different memory levels are influenced by a wide variety of factors. One example of this is that one type of CPU is usually made in different versions with different clock frequencies. Caches and main memory on the other hand have unchanged speeds. We therefore set out to measure this development by performing a number of measurements that can estimate the relative latencies on different computers.

1.4.1 Measuring Memory Latency

An analysis introduced by Spork [51] uses a refinement of the programming language *Pure C* [29], to measure the cost of Level 1 cache misses. Spork computes the cost of a cache miss, measured in the number of Pure-C instructions that could have been performed in the same amount of time. The result of the measurements are that for two architectures manufactured around 1998–99, the cost of a L1 cache miss is 10 Pure-C instructions. This measurement is made only for the penalty of a miss in the first level cache. As described above this penalty is expected to increase over time, and has changed since then. We are therefore interested in finding a method that determines the current status.

The introduction of super-scalar computers have, however, made the measurement of the penalty of fetching data difficult. This is because these types of CPUs are able to reorder the instructions, such that they can be performed as fast as possible. The result is that the CPUs can perform other computations while waiting for data to arrive from the memory hierarchy.

Optimizing compilers introduce a similar problem, as they may or may not reorder the instructions of a program, thus making it difficult to compare different programs. It is therefore not possible to measure the exact penalties simply by measuring the execution time of simple programs as Spork [51] did.

To measure the latencies in all the levels of the memory hierarchy, we have therefore used a tool called *lmbench* [37, 38]. LMbench is based on the same principles as Spork's program, but takes certain pitfalls into account. Furthermore, this is a commonly used software package. This tool is developed to give a wide range of performance measurements. We use the benchmarks for computing memory latencies, to measure each of the levels of the memory hierarchy. We do this on a a number of architectures in order to survey the development in memory technology.

1.4.2 Development in Memory Latency

The table in Figure 1.3 shows the results obtained for a number of computers which we have access to. This is meant as a rough survey to show trends in recent processor and memory technology.

It is important to remember, that when looking at these numbers, the superscalar computers are able to perform more than one instruction per cycle. This implies that if the Pentium 3 CPU is able to perform an average of 3 instructions

	Clock	Level 1	Cache	Level 2 (Cache	Main Mem.
Architecture	Freq. (Mhz)	Latency	Size	Latency	Size	latency
Digital Alpha EV56	493	2	8	18	96 ²	122
AMD Duron	807	3	64	23	64	133
AMD Athlon	1002	3	64	20	256	187
HP PA7200 1.1d	120	2	1024^{3}	-	-	41
HP PA8500 2.0w	551	3	1024^{3}	-	-	110
Intel Pentium	100	2	16	26	256	41
Intel Pentium 2	451	3	16	22	512	73
Intel Pentium 3	803	3	16	7	256	86
Intel Pentium 4	1480	2	8	19	256	262
Sun UltraSparc-I	167	3	16	9	512	46
Sun UltraSparc-IIi	333	2	16	10	2048	71
Sun UltraSparc-II ⁴	400	2	16	10	8192	127.8
Sun UltraSparc-III	750	2	64	13	8192	133

Figure 1.3: Latencies of cache and memory in number of clock cycles (rounded off to nearest cycle). The sizes of caches are in kilobytes.

per cycle the number of instructions performed during a level 2 cache miss is approximately $3 \times 86 = 258$. As it is difficult to write programs which after a load of a word have roughly 250 independent instructions, it is obvious that the minimization of cache misses is very important when developing fast programs.

Looking at the numbers in the table, two different developments in latencies are clear. As described earlier the general assumption has been that the relative latency of caches are increasing. This has, however, not been the case in general—Intel has actually been able to lower the latency of their level 2 caches. This development can be explained by two factors: Firstly, the projections of the development is based on the development in the speed of dynamic RAM, which is used for main memory. Caches, however, are made of faster types of memory, that are more similar to the technology used for CPUs, so that some of the improvements in CPU techniques can be transferred to the caches. Secondly, the hardware manufacturers have been able to improve the ways that caches are implemented. A major improvement has been achieved by moving the level 2 cache on to the same chip as the CPU.

Furthermore, the main memory in computers has evolved as expected, where the latency is significantly increased over time. This implies that whenever the problems are to large to fit into cache, the importance of reducing the number of misses greatly increases.

²The Digital Alpha EV56 has a unified level 2 cache, meaning that it is shared by both instructions and data. Furthermore it has 1MB off-chip level 3 cache, which is not included in this table.

³These architectures only have *one* level of cache. The PA7200 has also a small 2KB "assist" cache which together with the 1024KB form the total L1 cache.

⁴This computer has 24 CPUs and 24 Gb of main memory that can be accessed from all CPUs.

Due to this development the programmer must take the memory hierarchy into consideration when writing programs. Many algorithms can, though, be difficult for a programmer to adapt to efficient cache usage. In this case, the algorithm itself must be made to take the memory hierarchy into account. When designing algorithms it is, however, desirable to abstract from the low-level hardware issues and just concentrate on the algorithm itself. Algorithmic design models have therefore been developed, in order to facilitate this abstraction. These models try to offer a fairly simple model for designing algorithms. They must, however, also ensure that algorithms developed in the model, can be turned into implementations, which has a good performance in practice. In the following chapter we look into two of these.

1.5 Previous Work

At the point of writing we have only been able to locate five papers on cacheoblivious algorithms which touch on the subject of practical results.

A Master's Thesis by Prokop [44] has one small experimental result conducted on real hardware. It shows efficient behavior for a cache-oblivious Jacobi multipass filter, which executes in less than 70% of the running time of a standard RAM algorithm when the problem size is larger than the Level 2 cache size. Also, the update time per element is almost constant for the cache-oblivious implementation, whereas the RAM implementation slows down for each time data exceeds a memory level.

Frigo et al. [22] briefly show a few results on matrix-transposition and matrixmultiplication running on real computers. They consider the results preliminary, mainly because this is some of the first simple benchmarks published with cacheoblivious algorithms. However, the results clearly indicate that the cache-oblivious approach can be faster than the traditional iterative approach.

An article from 2001, Rahman et al. [45], study the influence of the translation look-aside buffer on the performance on the *dynamic predecessor problem*, using several variants of B-trees. The focus is to point out the importance of reducing misses in the TLB, as well as memory accesses. Different cache-aware solutions are implemented and compared to a cache-oblivious algorithm. The benchmarks are performed on real machines. Their results show that the cache-aware solutions are able to achieve better results in a memory hierarchy with one cache and a TLB, than the cache-oblivious implementation. They also show that cache obliviousness offers better theoretical and practical performance, than RAM-model algorithms which do not take TLB and other levels of the memory hierarchy into account.

Brodal & Fagerberg [15] implement different layouts of *search trees* and compare with benchmarks on real hardware. Their conclusion is that the good theoretical performance of cache-oblivious search trees are also noticeable in implementations. The results show that the cache-oblivious search tree achieve good performance, compared to standard RAM-algorithms for all but small data sizes and show robust behavior over several memory hierarchy levels. Also, the cacheoblivious search-tree implementation is able to compete with cache-aware variants.

Bender et al. [9] propose an alternative to B-trees [54] called *Dynamic Dictionary*. In order to compare the different implementations, they simulate a memory and compare the number of page faults found in their simulation, as well as the number of moves and comparisons needed. One of their goals is to investigate the influence of the sizes of the memory levels on the performance. Measurements are obtained using a simulated memory hierarchy to model which blocks are in memory and which are on disk. The simulated model uses full associativity and LRU replacement. The simulations show that their cache-oblivious search tree is able to compete with RAM algorithms. It should be noted, though, that the assumption of LRU replacement does not necessarily hold for real-life hardware as discussed in Section 2.3.2 on page 19. This would make it interesting to conduct similar benchmarks on a real memory hierarchy in order to confirm the results.

All in all, the results published so far indicate that a cache-oblivious approach seems to be able to give better performance than algorithms that are not optimized for memory hierarchies. On the other hand, it still seems that a cache-aware approach is faster, compared to the cache-oblivious approach. Still, no one has looked deeper into the subject on the actual relationship between theory and practice. How do contemporary computers conform to the assumptions made by the theory? Also, the amount of benchmarking on real computers is very limited and therefore an interesting subject to investigate further.

1.6 How to Read this Thesis

We here provide a quick overview of the content: Chapter 2 will introduce the field of cache-oblivious and external-memory algorithms. Subsequently, the chapter has an investigation of the differences between theory and the way contemporary computers work. If you are familiar with the cache-oblivious theory, you can skip most of this chapter, but we encourage you to read Section 2.3.6 on page 23, where we show that cache-oblivious algorithms automatically adapts to good use of the TLB and virtual memory system.

Chapter 3 treat two different priority queue algorithms which we have chosen as representatives of cache-oblivious algorithms. This chapter also discusses some details of these algorithms which the original articles do not cover.

In Chapter 4 we propose an algorithm for a optimal cache-oblivious priority deque, by making modifications to the Distribution Heap algorithm introduced in Section 3. Following that, Chapter 5 explains our general approach to conduct benchmarks, Chapter 6 explains details in regard to benchmarking the priority-queue algorithms, and the analysis of the results can be found in Chapter 7.

Finally, we sum up what our research have revealed about cache-oblivious algorithms and discuss practical relevance as well as suggest topics for future work on the subject in the conclusion in Chapter 8.

Chapter 2

Analyzing I/O Complexity

Say goodbye to all of this, and hello... to oblivion!

- Riff Raff, The Rocky Horror Picture Show

The development of CPU speed has—as discussed the previous chapter—lead to an increased importance of optimal use of the memory hierarchy. This is particularly important when the amount of data is too large for all of it to be stored in main memory, as the penalty of accessing disk is extremely high.

In the standard RAM model, the efficiency of an algorithm is measured by the number of instructions it incurs. Memory access is seen as having unit cost, regardless of where in the memory data is located. This model is not viable for reasoning accurately about modern computers with multiple levels of memory, because the introduction of a memory hierarchy implies different memory access costs depending on which part of the memory hierarchy data is stored in. If algorithms do not take the memory hierarchy into account, a severe performance penalty can be the result.

When the use of computers became more widespread in the 60's and 70's, magnetic tape drives were a common storage device compared to expensive hard disks. Computers at that time had a very limited memory and handling of large data sets had to involve temporarily reading and writing from/to tape which were slow compared to memory. Even then, it was realized that algorithms which were effective in main memory performed poorly on the external medium of tapes. Clever algorithms were developed to use the tape medium in an optimal way, by accessing data sequentially. An example of this is External Merge Sort as described by Knuth [30]. This is interesting because some of the same principles apply for algorithms on computers with a memory hierarchy.

The RAM model was developed because it was an easy, and understandable, abstraction of the hardware. As the RAM model no longer suffices for modern

computers, it becomes necessary to utilize more advanced models. The main tradeoff to keep in mind, though, is that a new model should both be a reasonable approximation to the actual hardware, as well as usable for both the developer of algorithms, and for the programmer who implements the algorithms. There is no use in cluttering the model with all kinds of details which become tedious to take into account when designing algorithms. Computers are becoming more and more complex as each component and its collaboration with the rest of the system is constantly being improved. The result is that a modern computer is filled with all kinds of caches and clever hardware logic which is *very* difficult to take into account. New models are therefore needed in order to develop efficient algorithms.

This chapter will describe the two models that try to incorporate the memory behavior of the algorithms in the analysis of algorithms: The *external-memory* model and the *cache-oblivious* model.

2.1 External-Memory Algorithms

When analyzing the memory behavior of algorithms the number of accesses to a level in the memory hierarchy is counted. In this field of research one access to a level in the memory hierarchy is often called one *I/O*, and several accesses are called *I/Os*.

The method of computing the number of I/Os needed to perform data processing in a two-level memory hierarchy, was described as early as 1972 by Floyd [21].

Another model that emerged as a result of the increasing memory latency, focused on the very high latencies in accessing data stored on hard disks. The basic idea of this model is that the complexity of an algorithm is analyzed by counting the number of disk accesses the algorithm needs. As the time needed to perform one disk I/O is so great, the number of instructions is irrelevant. This widely used model is therefore called the *external-memory* model or the *I/O* model. The model is formalized by Aggarwal and Vitter [2] in 1988, and Vitter and Shriver extended it in 1993 to handle parallel CPUs [55].

The model operates with two levels of memory: An internal and an external. The external complexity of an algorithm is measured as the number of I/Os performed on the external memory. For the analysis the model uses the following parameters:

N the number of elements being processed,

M the number of elements that fit into internal memory,

B the number of elements that fit into one block in internal memory, and

P the number of parallel disks¹.

 $^{^{-1}}P$ is often assumed to be 1 for simplicity, which we also do in this thesis.

It is assumed that $1 \leq B \leq M < N$, which ascertains that there is at least one cache block, and that the problem size is larger than the the size of internal memory, because otherwise it is not an external-memory problem. The efficiency of an algorithm is then measured as the number of memory transfers between the two levels of memory.

In some cases, it is still interesting to investigate the number of instructions in the algorithm, in the same way as in the RAM model— this is then called the *work complexity* of the algorithm.

Some bounds often used when analyzing the number of I/Os are:

- **Scanning** When a continuous part of data is scanned, like a large array, the amount of I/Os needed to perform this linear scan is $\Theta(\frac{N}{B})$ because this operation results in $\lceil \frac{N}{B} \rceil + 1$ compulsory cache misses. This bound is generally denoted scan(N).
- **Sorting** Aggarwal and Vitter [2] showed that under some assumptions, the asymptotic optimal number of I/Os needed to sort N elements is $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$. This bound on sorting is abbreviated sort(N).

Algorithms in the external-memory model are characterized by the fact that they are specifically tuned to the sizes of the memory hierarchy (M and B). The drawback of this is that external-memory implementations are highly platform dependent, as the sizes of caches vary between different types of computers. Also, the algorithms can be very complicated if there are more than two levels of memory involved.

The external-memory model used the term I/O to denote an access to the external memory, but when analyzing cache-oblivious algorithms, the term is used regardless of which level of the memory hierarchy is accessed.

A major point to consider when designing external-memory algorithms is that it is the responsibility of the programmer to control the movement of data blocks to and from external memory. This kind of control is possible, as the size of the different memories is known in advance and the programs can therefore be tuned to move blocks of the appropriate size. As the administration of memory can be a tedious task, different libraries have been developed, namely Duke University's TPIE [52], and an extension to the LEDA library [34] called LEDA-SM [35].

To illustrate how an external-memory algorithm works, we consider an algorithm performing a matrix multiplication of two $k \times k$ matrices. The algorithm [55] follows a divide-and-conquer method:

- 1. If the matrices are small enough to fit into internal memory (when $k \le \sqrt{M}$), perform the multiplication using any standard RAM algorithm, otherwise proceed as follows:
- 2. Divide the two matrices into eight sub matrices of roughly equal size:

$$A = egin{pmatrix} A_1 & A_2 \ A_3 & A_4 \end{pmatrix}, \ B = egin{pmatrix} B_1 & B_2 \ B_3 & B_4 \end{pmatrix}.$$

- 3. Reposition all eight new matrices into row-major order².
- 4. Compute the sub results C_1 , C_2 , C_3 and C_4 , by recursively using the algorithm:

$$C_1 = A_1B_1 + A_2B_3;$$

 $C_2 = A_1B_2 + A_2B_4;$
 $C_3 = A_3B_1 + A_4B_3;$
 $C_4 = A_3B_2 + A_4B_4;$

5. Reposition the sub results so that the result C is stored in row-major order:

$$C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}.$$

This algorithm uses the knowledge of the size of the internal memory to determine when to divide the problem further and when to perform the multiplication in internal memory.

It is very likely, that in order to minimize page faults, an external-memory algorithm will have to execute more instructions than an optimal RAM-algorithm. But, if we look at an example of the possible gain, it becomes clear that this is very affordable in some cases. A simple computation in [7], shows that this gain, when minimizing I/Os, can be the difference between a running time of 10 minutes for a sort(N) external-memory algorithm instead of 6 days for a corresponding O(N) RAM-algorithm.

Because of the simple yet powerful nature of the model, a large number of external-memory algorithms has been designed. Some recent surveys on the field are found in [5, 54].

2.2 Cache-Oblivious Algorithms

The main disadvantage of external-memory algorithms is that they are based on the knowledge of the memory structure and size, which makes it difficult to move implementations from one architecture to another. Another problem is that it is very difficult—if not impossible—to adapt some of these algorithms to work with multiple levels in the memory hierarchy.

The *cache-oblivious* model tries to alleviate this. The basis of the theory is that algorithms do not have any knowledge of the underlying memory structure. To illustrate this main difference, algorithms that are *not* cache-oblivious are called cache-aware (as, for example, external-memory algorithms).

²A matrix is stored in row-major order when all entries in a row are placed sequentially in memory. Normally this is done by an array of columns where each column itself is an array.

The idea of cache-obliviousness was introduced in 1999 by Prokop and his coauthors [22, 44]. A good survey of the results in the field, so far, has been made by Demaine [20]. The formal definition of cache-oblivious algorithms given by Prokop is:

An algorithm is *cache oblivious* if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses.

This definition implies that all RAM-model algorithms are also cache-oblivious algorithms. The large majority of RAM-model algorithms are, however, not optimal in their usage of the memory levels.

The principle of cache-oblivious algorithms is to make optimal use of each memory level, but without knowledge of the size of each level. This can, e.g., be achieved by a *divide-and-conquer* strategy, where the problem is recursively split into smaller parts. As an example, consider the matrix multiplication example above. The goal for a cache-oblivious version is to make sure that the sub matrices fit into internal memory, but *without* the knowledge of the size of the internal memory. The matrix example specified above can therefore be turned into a cache-oblivious algorithm if the divide step is continued until the sub matrices are some constant c in size, where c is chosen small enough to fit in the smallest level of memory on all known architectures. This way the algorithm will perform asymptotic optimal number of I/Os across different sized memories, because once a set of sub matrices fits into one cache, further iterations can be solved entirely within it [44].

Another difference, compared to external-memory algorithms, is that cacheoblivious algorithms assume that the moving of data between different levels in the memory hierarchy, is not the responsibility of the programmer. Instead it is performed by some automatic system being able to perform the *optimal* replacement of memory blocks. This might at first seem unrealistic, but we look into this later.

The fact that the algorithms must be without knowledge of the sizes of memory, has great implications. The problem of moving implementations between computers with different characteristics is removed, as the algorithm does not depend on knowledge about the computer. On the other hand, now the external-memory approach of dividing the problem into blocks of size B or M cannot be used—instead alternative methods and algorithms must be developed.

An interesting question is whether the cache-oblivious approach is better or worse than the cache-aware approach. In an article by Bilardi and Peserico [11], it is argued that in order for some sequences of instructions to be efficiently portable between computers with different memory systems modeled as HRAMs (hierarchical RAMs) [1], the program needs somehow to have knowledge of the memory parameters, to make it able to adapt to the actual memory hierarchy. Memory access time in the HRAM model is assumed to be a monotonic, nondecreasing function. In the extreme case, this function is either very steep (memory access

time increases rapidly when accessing distant levels in the memory hierarchy) or very flat (memory access time does not increase very much). Depending on which of these properties a certain machine has, it can be necessary to choose between storing sub results of a calculation in memory, or to recalculate. The argument is that, in order to determine the properties of a certain machine, the program needs to know the access time function and thus have knowledge of the memory hierarchy. This result implies that for some—but not all—sequences of instructions, it is impossible to achieve the same asymptotic bound in a cache-oblivious implementation on computers with sufficiently different memory hierarchies, as in a cache-aware implementation. This is a very theoretical result, though, and cacheobliviousness still seems to have a lot of potential in practice, which we are going to explore further.

2.3 The Ideal-Cache Model

To facilitate the design and analysis of cache-oblivious algorithms, a new memory model, called the *ideal-cache model*, was introduced in [44]. The main focus of this model is to make it simple to understand and thus easy to use when analyzing I/O complexity.

The model is made up of two levels of memory, namely a conceptually infinitely large memory and a data cache of M words divided into cache-lines of size B. The ideal-cache model is illustrated in Figure 2.1.



Figure 2.1: The ideal-cache model which consists of a small cache with ideal properties and a large main memory.

When analyzing an algorithm with an input of size N, two measures are used: The work complexity is the well-known measure of instructions in a RAM-model, while the cache complexity or I/O complexity represents the number of cache misses in a cache of size M, with block size B. Some of the concepts (e.g., sort(N) and scan(N)) from external memory are also used, when analyzing algorithms in the ideal-cache model. As in external-memory algorithms, the values of M and B are unknown when the algorithm is designed. The difference is that when implementing a cache-oblivious algorithm, these values are not used in any way. Only when analyzing cache-oblivious algorithms, M and B are still used—even though they are unknown to the implementation.

The main idea behind the ideal-cache model, is to use it for algorithmic design when finding upper and lower bounds of algorithms—due to its simplicity. The upper bound analysis is straightforward, because no matter which replacement strategy is used in real–life, the optimal replacement strategy will always perform at least as well, and thus we can find an upper bound on the number of cache misses simply by choosing one possible replacement sequence, which makes it a valid, but not necessarily tight, upper bound.

Another important point to observe, is that the CPU complexity of an algorithm is an upper bound on the cache complexity of the algorithm. This is realized by the simple observation that a cache miss can only follow as a result of a load or store operation, so the number of cache misses can never be larger than the number of operations made by the algorithm.

Computing the lower bound is a bit trickier because it must be proved that the optimal replacement strategy is asymptotically comparable to the actual replacement strategy used.

Nevertheless, the ideal-cache model is a very usable tool for analyzing algorithms. The desired property of a simple model is achieved by using an abstraction of the computer, which is as simple and "optimistic" as possible, and the model is therefore based on a couple of assumptions, that need further explanation. The remaining part of this section covers different aspects of the ideal-cache model.

Finally, it is important that the results achieved using the ideal-cache model are comparable to real-life results. The relationship between results achieved in the model, and in reality, should therefore be properly described. It is also desirable that bounds for the ideal-cache model should be comparable to the corresponding bounds for the external-memory model, as this would make it possible to compare algorithms from the different design paradigms.

2.3.1 Tall Cache

One of the assumptions of the ideal-cache model is that the cache is "tall", that is:

$$M = \Omega(B^2).$$

This guarantees that the block size is not wider than the total number of blocks. This is useful for simplifying the computations of I/O complexity. This assumption can in some cases—as described by Demaine [20]—be weakened to $M = \Omega(B^{1+\gamma})$, where $1 > \gamma > 0$.

As an example, consider matrix multiplication where the tall cache assumption guarantees that the cache can hold a square matrix of size $O(B) \times O(B)$. This assumption is very realistic, as a larger *B* results in a larger cache-miss penalty [25, p. 426]. This is the reason why tall caches are found in all contemporary architectures.

The table in Figure 2.2 shows the properties of B on contemporary architectures. The results indicate that in practice the constant in the $M = \Omega(B^2)$ term for caches is between 2–1024 and the tall cache assumption is therefore realistic.

2.3.2 Optimal Replacement

It is fairly simple to analyze the I/O complexity in the ideal-cache model, under the assumption that the memory uses an automatic optimal replacement policy. This is a major difference between the ideal model and real life. When using the ideal-cache model, it is assumed that the replacement policy is the optimal offline algorithm. It is, however, not possible to implement this in a computer because it is impossible to acquire knowledge of the future behavior of a program. It can be shown, though, that it is fair to use the ideal-cache model anyway, under the assumption that the size of a real life least-recently-used cache is larger than the cache in the ideal-cache model.

When analyzing the cache complexity in a cache that uses optimal replacement policy, the important observation is that an LRU replacement policy only adds a constant overhead on the complexity if the LRU replacement policy is used in a memory larger that the one used by the optimal policy.

This relation was proven in a classic paper by Sleator and Tarjan [50]. They proved that for any online replacement policy, *A*, there exists a sequence of cache accesses that gives the worst-case number of cache-misses

$$Q_A \ge \frac{M_A}{M_A - M_{OPT} + 1} Q_{OPT},$$

where M_A is the size of the cache that A uses, M_{OPT} is the size of the cache used for the optimal replacement, and Q_{OPT} is the cache misses for the algorithm, when using the optimal cache. Here it is assumed that $M_A \ge M_{OPT}$.

They also prove that the number of cache misses for a cache using LRU-replacement strategy Q_{LRU} is upper bounded by:

$$Q_{LRU} \le \frac{M_{LRU}}{M_{LRU} - M_{OPT} + 1} Q_{OPT}.$$

This implies that when the two replacement policies are implemented in caches of the same size ($M_{LRU} = M_{OPT} = M$), there exists a worst-case access pattern, where LRU will perform *much* worse than the optimal replacement policy:

		M (KB)	B (bytes)	Maximal constant in Tall Cache Assumption	Associativity (x-way)
	Alpha EV56	8	32	8	1
	AMD Duron	64	64	16	2
	AMD Athlon	64	64	16	2
	HP PA7200 1.1d ³	1024	32	1024	1
che	HP PA8500 2.0w ⁴	1024	32/64	1024/256	4
cac	Intel Pentium	16	32	16	2
11	Intel Pentium 2	16	32	16	4
eve	Intel Pentium 3	16	32	16	4
Ľ	Intel Pentium 4	8	64	2	4
	Sun UltraSparc-I	16	32	16	1
	Sun UltraSparc-IIi	16	32	16	1
	Sun UltraSparc-II	16	32	16	1
	Sun UltraSparc-III	64	32	64	4
	Alpha EV56 ⁵	32/64	96	96/24	3
	AMD Duron	64	64	16	16
	AMD Athlon	256	64	64	16
he	Intel Pentium	256	32	256	2
cac	Intel Pentium 2	512	32	512	4
2	Intel Pentium 3	256	32	256	8
svel	Intel Pentium 4	256	128	16	8
Ľ	Sun UltraSparc-I	512	64	128	1
	Sun UltraSparc-IIi	2048	64	512	1
	Sun UltraSparc-II	8192	64	2048	1
	Sun UltraSparc-III	8192	512	32	1

Figure 2.2: Table of associativity and cache sizes on real hardware.

$$Q_{LRU} = \Theta\left(\frac{M}{M - M + 1}Q_{OPT}\right) = \Theta\left(M \times Q_{OPT}\right).$$

However, with the assumption that the LRU cache is larger that the ideal cache, the LRU replacement policy is only a constant factor worse than the optimal replace-

³The PA7200 also has a 2KB "assist cache" with full associativity which together with the 1024KB form the total level 1 cache.

⁴The PA8500 supports both 32 and 64 bytes line size.

⁵The Digital Alpha EV56 has a unified level 2 cache, meaning that it is shared by both instructions and data. In addition it has a direct-mapped 2MB level 3 cache with a cache-line size of 32/64 bytes.

ment policy. For example, if the LRU cache is twice the size of the optimal cache, the LRU cache will in the worst case imply twice the number of cache misses:

$$Q_{LRU} \le \frac{2M_{OPT}}{2M_{OPT} - M_{OPT} + 1} Q_{OPT} < 2 \times Q_{OPT}.$$

Generalizing this: A LRU cache of size q is in worst case only a factor c worse that a optimal cache of size $\left(1 - \frac{1}{c}\right)q$ [50].

On the basis of this, Prokop [44] defines a cache complexity to be *regular* when the cache complexity of an algorithm run in a cache of size M is asymptotically equal to the cache complexity of the same algorithm when run in a memory with a cache of size 2M.

Based on the proofs mentioned above, it is clear that any algorithm that has a regular bound will have the same asymptotic complexity in a larger LRU cache. As a cache-oblivious algorithm does not have any knowledge of the sizes of caches, it is reasonable to make this assumption and to argue that bounds found in the ideal-cache model are comparable to real life.

It should, however, be noted that in practice LRU is often approximated, in order to make the replacement algorithm simpler and faster. In the virtual memory part of an operating system there can be several other things to be considered like speed (the kernel functions should be fast), space (not too much extra space should be used for time stamping) and I/O (it is desirable to move big contiguous blocks of memory to disk because this makes better use of the disk I/O-system. This potentially makes the difference between the ideal-cache and practice bigger.

2.3.3 Automatic Replacement

A major difference between the external-memory I/O-model and the ideal-cache model is the assumption of automatic replacement. Where the I/O-model leaves it to the programmer to handle block transfers to and from disk, this is impossible for a cache-oblivious program, because of the lack of information about block sizes. This assumption is backed by the fact that a cache by nature implies automatic replacement. An interesting question is whether cache-oblivious algorithms are comparable with external-memory algorithms in spite of this difference.

A cache-oblivious algorithm could be run on top of some cache simulation software and in practice use memory in an external-memory fashion. The software simulating the cache will then handle the moving of blocks to and from main memory following a LRU strategy. This software must offer an O(1) access-time to any data currently present in the simulated memory and guarantee that the I/O cost of the operations are identical to the ones that would have occurred in actual hardware. That is, any access to a word currently in the simulated cache must be accessed without causing any I/Os, whereas for a word not currently in it may result in one I/O. This simulation software could then be used to convert any algorithm designed for the ideal-cache model into an external-memory algorithm. Such a cache simulation software could be achieved by using a 2-universal hash function to hold the location of the cache lines in the primary memory [18, 44]. The disadvantage is that the hash function only gives an *expected* bound of O(1) on the complexity of finding an element that is located in the simulated cache. If this property can be accepted, though, it is possible to compare algorithms in the external-memory and cache-oblivious models, which is desirable. Furthermore, this assumption has the implication that a lower bound in the external-memory model also is a lower bound in the ideal-cache model.

2.3.4 Associativity

The assumption of full associativity is not necessarily realistic when dealing with real-life computers, as hardware caches have limited associativity due to the high cost of implementing full associativity with regard to both hit and miss latency as well as production price [25]. In the case where pages from the disk is cached in main memory, the cache functionality is implemented in software, and it is therefore not a problem to have full associativity in this situation.

The problem with limited associativity is that if the cache does not use full associativity, it is possible to construct a stream of requests which will result in a much higher number of I/Os than would have been needed in a fully associative cache.

The main point, in regard to the cache-oblivious theory, is that the property of full associativity is chosen because it is simple to deal with. Also, the requirement for full associativity is not frequently used in proofs of cache complexity, and so for most algorithms this is not a problem. For algorithms which rely heavily on full associativity the ideal-cache model will yield better bounds than can be obtained in practice.

See Figure 2.2 on page 20 for a list of hardware cache associativity on contemporary computers. The results indicate that 2, 4, or 8-way set associative caches are common, but direct-mapped caches are also used in some architectures.

2.3.5 Multiple Levels of Memory

Another interesting difference between the ideal-cache model and the hardware that exists in real computers, is that the ideal-cache model only incorporate two levels of memory, while computers today typically have 3–5 levels. To be able to use the theoretical results achieved, when studying algorithms that use the ideal cache model, it must be proved that we can generalize the results to hold for computers with multiple levels of memory.

Because an algorithm in the ideal-cache model does not have knowledge about the dimensions of the cache, an optimal algorithm in the ideal-cache model will incur an optimal number of I/Os in each of the levels in a multi-level memory hierarchy. It is furthermore fair to assume that the different levels of cache in a multi-level cache hierarchy are inclusive [25, p. 616]. These two facts ensure that an algorithm which is I/O optimal in the ideal-cache model, is also I/O optimal on each of the levels in the multi-level memory hierarchy. As the sum of optimal terms is itself automatically optimal, this ensures that I/O optimal algorithms in the ideal-cache model are also I/O optimal in memory hierarchies with multiple levels.

The result of the analysis will therefore be valid between each of the boundaries in the memory hierarchy: An algorithm which is optimal in the ideal-cache model is, therefore, also optimal in a corresponding multilevel ideal-cache model.

We have illustrated this property in Figure 2.3 which shows the repeated use of the analysis in a multilevel memory hierarchy.



Figure 2.3: The multilevel ideal cache model

2.3.6 The Virtual-Memory System

As described in Section 1.3 on page 5 the memory hierarchy in current computers has a virtual memory system that also has an impact on the time it takes to fetch data from the memory hierarchy. It is therefore interesting to discuss whether the ideal-cache model and the cache-oblivious theory are able to incorporate the virtual-memory system in the analysis. This subject has not been treated much in the literature, although Rahman, Cole and Raman [45] did give preliminary experimental results indicating the importance of optimal TLB usage, which showed that cache-oblivious algorithms *will* result in optimal usage of the TLB.

When computing the total cost of an I/O, the costs of the misses in the ordinary caches, and the TLB, are added as seen by the simplification in the following equation: data fetch time = total cost in the ordinary memory hierarchy + total cost of the address conversion.

Based on this formula, a natural abstraction is to consider the complete memory system as two parallel autonomous memory hierarchies. In this abstraction any memory access requires an access to both memory hierarchies and might result in misses in one or both of the hierarchies. This abstraction of the memory system is illustrated in Figure 2.4.



Figure 2.4: Abstraction of the virtual memory system: The well known cache hierarchy is depicted in the bottom half and the TLB hierarchy in the upper half of the figure. The combination of these two hierarchies illustrates a memory system.

This abstraction sees the memory system as two distinct memory hierarchies, where the total cost of the memory accesses are the sum of the costs from each of the memory hierarchies. The ideal-cache model was invented to describe the normal memory hierarchy (as illustrated in Figure 2.1 on page 17). It can, however, also describe the TLB-hierarchy: The TLB can be modeled in the ideal-cache model as a cache with a block size of 1 page, using full associativity and optimal replacement policy. Even though the TLB only holds one entry in each block, this entry is a pointer to a whole page frame and can therefore be used to translate a range of virtual addresses making the effective block size equal to a page.

These observations naturally lead us to a generalization of the ideal-cache model to work in computers with a virtual memory system. The argument is that if an algorithm is optimal in the ideal-cache model it will be optimal on each of the levels in a memory hierarchy. This argument is valid for both the cache hierarchy and for the TLB hierarchy on Figure 2.4. As the total cost of the memory system is the sum of these two, the total cost will also be optimal because the sum of two optimal costs will itself be optimal.

Another important aspect of the virtual-memory system which could be a problem is the way memory is allocated in virtual and physical memory. If a contiguous chunk of memory is allocated by a program, this is done in virtual memory address space. When this memory is accessed the virtual memory addresses are converted to physical addresses by the memory management unit (MMU), which uses the TLB to cache frequently used addresses. The memory is divided into pages, which can be present in main memory or swapped to disk, but there is no guarantee that the pages are written contiguously on disk—in the worst case, they could be scattered around the disk.

The result of this mapping might be that contiguous virtual pages are actually stored in noncontiguous physical pages. Fortunately, this will not influence the ideal-cache assumption as a page itself always will be allocated as one contiguous block in physical memory, as the pages are simultaneously used as the blocks in main memory.

The virtual memory system makes the memory hierarchy more complex and results in a number of extra details that one could suspect might add extra asymptotic cost on fetching data. Luckily, the ideal-cache theory is able to incorporate these extra details and an I/O-optimal algorithm in the cache-oblivious model is therefore also optimal in a virtual memory system.

2.4 An Introduction to Cache-Oblivious Techniques

A number of techniques can be used to develop cache-oblivious algorithms and data structures. The majority of all cache-oblivious results uses one or more of these techniques, so here we give a short introduction to these:

Sequential Data Access The obvious approach is to make an algorithm process data in a sequential way. This will use the memory hierarchy optimally as N elements can be processed with $O\left(\frac{N}{B}\right)$ cache misses. An example of this is *scanning* an array to find the maximum element.

- **Divide-and-Conquer** The cache-aware approach for utilizing the cache efficiently is to *block* the problem. That is, dividing it into subproblems of a size suitable to be in the cache. The cache-oblivious counterpart is to use a divide-and-conquer approach. The idea is, that once a subproblem fits into a level in the memory hierarchy the further dividing down to a constant size will not cause any further cache misses, and is therefore cheap. An example of this approach is the matrix-multiplication described in Sections 2.1 and 2.2.
- **Recursive Layout** For a static data structure—like, for example, a tree—data can be placed in memory in such a way that it improves the usage of the memory hierarchy. One example is to use a recursive layout called the *van Emde Boas layout*. Conceptually, the tree is laid out in memory, by splitting the tree in a top subtree, and a number of bottom subtrees, so that the height of all these subtrees are roughly half the hight of the original tree. These subtrees are now placed consecutively in memory, so that the top subtree is placed first, followed by the bottom subtrees. The layout of each of the subtrees is determined recursively. An example of this layout is shown in Figure 2.5.

The advantage with this layout is that paths from root to leaf-nodes are *blocked*, so that a number of nodes on the path will be placed in the same memory block, independently of the size of the block. Note that the algorithm traversing the tree does not have to take on a divide-and-conquer approach, because the data layout makes sure to improve the use of the memory hierarchy. See [15, 22] for examples of this technique.



Figure 2.5: The recursive van Emde Boas layout of a binary tree. The dashed lines show the recursive division in subtrees, that are recursively laid out in memory in consecutive blocks. The numbers in the nodes correspond to the layout.

Lazy Evaluation using Buffers Another approach, for achieving sequential data processing, is by using *buffers*. This technique is described by Arge, and used to develop the cache-aware *buffer-tree* data structure [4]. The idea can, however, be reused in a cache-oblivious context, if the size of the buffers are not fixed to the sizes of the actual memory hierarchy, but instead grow, beginning with a small constant size. This way, the buffers will at some point fit into a cache level, and processing a buffer of this size will not incur more cache misses on that particular level.

The advantage of this technique is that the *work* is buffered in a lazy fashion. Only when the buffer is full, the content of a buffer is moved to the next larger level, and the elements stored in the buffer are all processed in one operation. This way, the main data processing occurs on the smallest level, and elements are moved up or down in the structure in a lazy—or batched—fashion. The cache misses incurred can be amortized between the elements in the buffer. Because whole chunks of elements are moved from one buffer to another, data can be processed sequentially. This technique is for example used by the priority-queue data structures, which we treat in Chapter 3.

Chapter 3

Cache-Oblivious Priority Queues

Everything that can be invented has been invented.

- Charles H. Duell, Commissioner, U.S. Office of Patents, 1899.

This chapter presents two cache-oblivious priority queue algorithms as well as our considerations and changes needed to make implementations of them. A priority queue is a data structure that supports the following operations:

- **Insert**(\mathbf{x}) adds the element x to the data structure.
- **Extract**() returns the *smallest* element currently stored in the data structure and removes it, where "smallest" is determined by a specified *ordering relation*. This way it is possible to change the ordering of the elements.
- Some priority-queue algorithms offer additional operations like:
- **Delete(x)** deletes an element with priority x from the data structure.
- **Construct(X)** constructs a priority queue containing all elements in the sequence X. This is similar to repeatedly calling Insert() on each of the elements, but is offered when a faster method is available for constructing an initial state of the data structure, with a given set of input data.

In the field of external-memory algorithms, several priority queues have been developed [54], but as to the nature of external-memory algorithms, these depend heavily on the knowledge of memory and block size. To design a cache-oblivious priority queue, other approaches are needed. So far, two different algorithms have been developed, both with optimal amortized bounds on insertion and extraction of elements. Two different solutions have been suggested—both published in 2002. The first was introduced by Lars Arge et al. [6] and the second by Gerth S. Brodal & Rolf Fagerberg [14].

The data structure by Brodal & Fagerberg is based on a cache-oblivious *merger* introduced by Prokop [44] and which was originally used for sorting. This construction is called a *funnel*, and therefore the authors named their priority queue the *Funnel Heap*. Arge et al. did not name their priority queue, but as it uses the opposite approach, where elements are distributed instead of merged, we name this the *Distribution Heap*. These names are perhaps a bit misleading, as the term "heap" is normally seen as being a specific RAM-based solution to the problem of making a priority queue data structure [19]. We therefore use the term "heap" as a name and the term "priority queue" when talking about the data structure itself. The following sections describe the theory, as well as the practical implementation details of the two data structures.

3.1 The Distribution Heap

The Distribution Heap is suggested by Arge et al. [6] for developing cache-oblivious algorithms for graph problems.

The data structure is made in a top-down fashion with levels of decreasing size. The size of the largest level is chosen to N. The size decreases by the power of 2/3 for each level. When the size of the next level would have been of size smaller than some constant c, the recursion is stopped. The total number of elements in the structure is N and there are $\Theta(\log \log N)$ number of levels in total¹.

A level is named after its size, so that a level of size X is called level X. The levels from the largest to the smallest are thus *named*: $N, \ldots, X^{3/2}, X, X^{2/3}, X^{4/9}, \ldots$

The main idea, for achieving cache obliviousness, is to work on *buffers*: Each level consists of a number of *down buffers* and an *up buffer*, as illustrated in Figure 3.1. On a level X there are $X^{1/3}$ down buffers, each of size $2X^{2/3}$, as well as one up buffer of size X. Level X can therefore hold up to 3X elements. Note also, that the size of one down buffer on level X is twice the size of the up buffer one level down (level $X^{2/3}$).

Intuitively, the idea behind the Distribution Heap is that each down buffer contains an interval of elements. The buffers containing the smallest elements are found in the smallest level. The up buffers contain elements that have not yet been placed in the right interval and therefore are on their way up to the levels above to find the down buffer that represents the interval containing the element. An example showing how a Distribution heap might look is found in Figure 3.2.

¹This top-down recursive definition of the level sizes is well hidden within the article by Arge et al. [6], but after some e-mail correspondence with the authors we confirmed that this is the way the data structure was meant to work.



Figure 3.1: The layout of a Distribution Heap [6].

61 99			
15 25 15	28 29	50 50 60	empty
15 57 99	32		
152	12 12 14	13	

Figure 3.2: Example illustrating a Distribution Heap.
Formally, the Distribution Heap maintains three invariants:

At any level, the elements are sorted *among* the down buffers, so that each element in a down buffer is smaller than any element in the next down buffer. That is, for any down buffer a on level X (called d^X_a) where a ∈ {1, 2, ..., X^{1/3} − 1} any element f in d^X_a is smaller than any element g in d^X_{a+1}:

$$\forall f \in d_a^X, \ \forall g \in d_{a+1}^X : f < g.$$

2. Any element f, in a down buffer on level X is smaller than any element g in the up buffer u^X on the same level. That is, for any $a \in \{1, 2, ..., X^{1/3}\}$:

$$\forall \ f \in d_a^X, \ \ \forall \ g \in u^X : f < g.$$

3. Any element f, in a down buffer on level X is smaller than any element g in a down buffer in the level above $(X^{3/2})$. That is, for $a \in \{1, 2, ..., X^{1/3}\}$ and $b \in \{1, 2, ..., X^{1/2}\}$:

$$\forall f \in d_a^X, \ \forall g \in d_b^{X^{3/2}} : f < g$$

Furthermore, it is required that each down buffer on level X contains at least $\frac{1}{2}X^{2/3}$ elements. As the size of each down buffer is $2X^{2/3}$, this corresponds to keeping the buffers at least 1/4 full.

The largest element in a down buffer marks the boundary between two down buffers, and is called the *pivot* element. It is stored in the buffer in such a way that it can be retrieved using O(1) I/Os. A down buffer contains elements that are in the interval between the pivot element from the previous down buffer, and the priority of the down buffer's own pivot element. Elements inside one down buffer are not necessarily sorted among each other.

Intuitively, the invariants described above provide a structure, where the down buffers hold the smallest elements, and where the down buffers on each level contain elements smaller than on the levels above it. The elements in the up buffers, on the other hand, are elements that have not yet found their place in an appropriate down buffer. The only fact known about them, is that they are larger than any element in the down buffers on the current level, and therefore also larger than elements in the down buffers in all levels below. In other words, we know that they have to move *up* to higher levels to find their place in a suitable down buffer.

Elements are moved up and down the tree by means of two basic operations: *push* and *pull*. Arge et al. describe these two functions, but there are details that need to be considered when the Distribution Heap is to be implemented in practice. These will be covered in Sections 3.1.5 to 3.1.11 when we will propose solutions to these extra details.

Because we need to extend the original operations, we do not present pseudocode until all the extra details have been discussed. The pseudo-code for the modified operations can be found on pages 45–47.

3.1.1 Push

Push moves the elements in the up buffer on level X into the buffers on level $X^{3/2}$. This operation is needed when the up buffer on level X is full. A push involves several steps:

First the up buffer—of size X—is sorted. The sorted elements are then distributed among the down buffers on level $X^{3/2}$. The distribution starts by trying to insert the first element in the first down buffer. If the element is smaller than the pivot element of the first down buffer, it is inserted in this buffer. If, however, the element is larger than the pivot element, the operation proceeds to the next down buffer in order to determine whether the element fits there. This procedure is repeated for all elements and it ensures that the elements in the down buffers continue to maintain Invariant 1.

When the operation has tried all down buffers, the remaining elements are inserted in the up buffer on the same level $(X^{3/2})$. Note that when this occurs, the remaining elements are all larger than all elements in all down buffers, which ensures that Invariant 2 is maintained.

A down buffer might become full during a distribution step, which means that it cannot contain all elements that are smaller than its pivot element. A full down buffer on level $X^{3/2}$ contains 2X elements, and these are now split into two down buffers, each containing X elements. This is done by finding the median of the elements cache obliviously (see Section 3.1.9 on page 42) and then partitioning the elements into two buffers.

This split of a buffer results in two buffers: The buffer that was split which now contains the smaller half of the elements, and a new buffer containing the larger elements. To maintain Invariant 1, the down buffers are reordered. To allow this, the buffers are stored in a way that allow changing their order using a linear number of I/Os—e.g. by a linked list. Figure 3.3 shows an example of a change of the ordering of the down buffers.

Each level has a maximum number of down buffers, so if all down buffers are already used, one down buffer needs to be emptied before the full buffer can be split. To maintain invariants 1 and 2, the down buffer which contains the largest elements is emptied by moving its elements into the up buffer.

The up buffer on level $X^{3/2}$ might also become full. In this case the up buffer is recursively pushed to the next level above (level $X^{9/4}$).

I/O Complexity of Push

This section describes the complexity of pushing X elements from level X into level $X^{3/2}$. We will go through the entire analysis of the Distribution Heap and



Figure 3.3: Down buffers are stored in a structure where it is possible to change their order. Before the split, three down buffers i, j and k contain elements, and there is one empty down buffer. After the split of j, half of the elements are copied to the empty buffer l. The ordering of the buffers is now: i, j, l, k.

clarify details that are not obvious from the analysis by Arge et al. [6]. We do this as we need the analysis in Chapter 4 when we develop a new cache-oblivious algorithm based on the Distribution Heap.

If an optimal cache-oblivious sorting method is used to sort the up buffer, it can be sorted using sort(X) I/Os.

To distribute the elements, a scan of the sorted elements is needed. This can be done using scan(X) I/Os, but there is also the cost involved of visiting each of the down buffers: Even though the distribution does not necessarily need to move elements to all buffers, at least the pivot element of each buffer needs to be read for each of the $X^{1/2}$ down buffers on level $X^{3/2}$. The total cost of distribution is thus $scan(X) + X^{1/2}$ I/Os.

If a down buffer becomes full, a split is performed. A split can easily be done by first finding the median cache obliviously, using scan(X) I/Os, and then scanning the full down buffer, and the two new down buffers, yielding scan(X) I/Os. It is important to note that these split operations only take place each time X elements have been inserted in a down buffer on level $X^{3/2}$. The amortized cost per element of a split is therefore

$$\frac{scan(X)}{X} = O\left(\frac{1}{B}\right).$$

If the up buffer on level $X^{3/2}$ becomes full, the $X^{3/2}$ elements it contains are recursively pushed into level $X^{9/4}$. Hereafter $X^{3/2}$ elements must be inserted on level $X^{3/2}$, before another recursive push is needed.

Ignoring the cost of the recursive pushes, the total I/O complexity of a single push is $scan(X) + sort(X) + X^{1/2}$ plus an amortized cost per element of O(1/B), for the splitting. In Section 3.1.3 we use this complexity of one push to one level to compute the total complexity of a sequence of operations.

3.1.2 Pull

Pull is used when there are not enough elements in the down buffers on level X. This is fixed by extracting X elements from the level above $(X^{3/2})$ and inserting them into level X.

Pull start by handling the case where the down buffers on level $X^{3/2}$ contain less than $\frac{3}{2}X$ elements. This is necessary, because we need enough elements left on level $X^{3/2}$ to make at least one down buffer containing $\frac{1}{2}X$ elements after the pull. This is fixed by recursively pulling $X^{3/2}$ elements from level $X^{9/4}$ before we proceed with this pull operation. The recursive pull will insert $X^{3/2}$ elements on level $X^{3/2}$, which ensures that level $X^{3/2}$ now contains more than the $\frac{3}{2}X$ elements we need.

We now know, that the down buffers on level $X^{3/2}$ contains at least $\frac{3}{2}X$ elements. The X smallest elements are removed from level $X^{3/2}$ by sorting each of the first three down buffers² and extracting the X smallest elements. Invariant 1 ensure that we can concatenate sorted down buffers, to form a sorted sequence.

We need to use three down buffers, as each of them contains at least $\frac{1}{2}X$ elements. Using three down buffers guarantees that the sorted sequence contains at least $\frac{3}{2}X$ and at most 6X elements. After removing X elements this ensures that there remains between $\frac{1}{2}X$ and 5 elements. These can then form one, two, or three new down buffers, while still fulfilling the requirement, that all down buffers must contain at least $\frac{1}{2}X$ elements.

The X elements that were extracted from level $X^{3/2}$, must now be inserted in the buffers on level X. The elements in the up buffer on level X are sorted and merged with the extracted elements to form the sorted sequence α . This is necessary as the invariants do not ensure any ordering between the elements in α . The up buffer is refilled with the same number of elements as before, by taking the largest elements from α . This ensures that the elements now in the up buffer cannot be smaller than the elements that where there before, and thus maintains Invariant 2. The remaining elements in α are divided into sections of size $O(X^{2/3})$ and inserted in empty down buffers on level X.

All in all, a pull adds X elements to the down buffers on level X.

I/O Complexity of Pull

The I/O complexity of a pull consists of two parts: The cost of a pull when there are enough elements in the down buffers, and the cost when there are not enough elements and a recursive pull is needed. In the following we ignore the cost of the recursive pulls, as this is included in the complexity analysis of insert and extract, which we perform in the next section.

²This is actually only possible if we assume that X is larger than some constant. This is taken into account in Section 3.1.7 on page 41 when we compute this limitation of the size of the smallest level.

The X smallest elements are pulled from the down buffers on level $X^{3/2}$. These elements are removed by first sorting the elements in the first three down buffers and then reinserting the remaining elements in the down buffers using a scan. As three down buffers can hold at most 6X elements. The cost of this procedure is thus scan(X) + sort(X) I/Os.

To insert these X elements on level X, we first sort the O(X) elements in the up buffer using sort(X) I/Os. We then merge the two times O(X) elements, by scanning them in scan(X) I/Os. Finally we use scan(X) I/Os to move the elements into the up and down buffers. The total I/O complexity of the insertion of the X elements is therefore sort(X) + scan(X).

Because X elements are inserted on level X during the pull, the same amount of elements have to be pulled from this level, before another recursive pull needs to be performed. The cost of the pull can therefore be amortized over the X elements.

Still ignoring the cost of the recursive pulls, the total amortized cost of pulling one element from level $X^{3/2}$ is

$$O\left(\frac{1}{B}\log_{M/B}\left(\frac{X}{B}\right)\right) + O\left(\frac{1}{B}\right).$$

3.1.3 Insert and Extract

The description of the Distribution Heap does not mention how inserts and extracts of individual elements are handled. Let the size of the smallest level in the priority queue be called d. This level is in the range between $c^{2/3}$ and c, and the description and analysis of push and pull is based on the assumption that the operations are performed on a number of elements each time. A pull or push on level d is designed to work on $d^{2/3}$ elements. To handle this we have added an *insertion buffer* and an *extraction buffer*—each of size $d^{2/3}$. Both buffers are maintained in sorted order which does not influence the complexity, because d is relatively small.

Whenever an element is inserted into the priority queue, it is first inserted into the insertion buffer. When this buffer is full, the contents of the insertion buffer and the extraction buffer are merged to form a single sorted stream. The largest $d^{2/3}$ elements are now inserted into the priority queue with a push, and the remaining elements are inserted into the extraction buffer. Note that this ensures that the largest elements from the two buffers are inserted in the priority queue.

The extraction buffer is used to store elements pulled from the bottom level of the Distribution Heap. To extract one element, we first make sure that the extraction buffer is not empty—the smallest element is then found in either the insertion or the extraction buffer.

The number of elements contained in the extraction buffer is unchanged by a push of elements. This implies that, when elements have been pushed into the priority queue, $d^{2/3}$ elements have to be inserted before the next push occurs.

In the description above we assume that the inserting and extraction buffer is implemented as a simple buffer. As we need to be able to insert new elements and the extract the largest elements from these "buffers", it is natural to consider using small priority queues instead.

We have implemented the Distribution Heap so that it is possible to use both versions. Experiments has shown that the simple buffers is up to 40% faster then a small priority queue, so we use the simple version in all the benchmarks we perform in this thesis.

I/O complexity of Insert and Extract

In Section 3.1.1 on page 32 we showed that the cost of one push of X elements to level $X^{3/2}$ uses $scan(X) + sort(X) + X^{1/2}$ I/Os. We would, however, like to remove the $X^{1/2}$ and the scan(X) terms, in order to get the optimal cache-oblivious bound.

We first describe how to remove the scan(X) term. There is a constant number of levels where X < cM for some suitable constant 0 < c < 1. These entire levels can be kept in memory, because they only use a fraction of M. In this case, the scanning can be done without accessing memory. For the remaining levels $(X \ge cM)$, we know that the logarithmic term $\log_{M/B} \frac{X}{B}$, which is the difference between scan(X) and sort(X), is $\Omega(1)$. This implies that the scan(X) term is dominated by the sort(X) term.

We now need to remove the $X^{1/2}$ term. This can be done by analyzing a sequence of N/2 insert and extract operations: After N/2 operations the priority queue is rebuild as will be described in Section 3.1.6 on page 40. This guarantees that at least X elements have to be pushed into level X before another recursive push is performed, because the up buffer on this level can hold exactly X elements and this is empty after a rebuild. Likewise, X elements have to be pulled from level X to level $X^{2/3}$ before a recursive pull is needed. This happens when there is only one down buffer left and this buffer contains less than $\frac{1}{2}X^{2/3}$ elements. After a rebuild all down buffers are half full and therefore contains X elements altogether. For each pull $X^{2/3}$ elements are moved to level $X^{2/3}$. This must be done exactly $X^{1/3}$ times before X elements have been pulled from level X and a recursive pull is need, because after $X^{1/3} - 1$ times there are still $X^{2/3}$ elements left in the last remaining down buffer.

During the N/2 operations O(N/X) pushes are performed on level $X^{3/2}$, as one push to level $X^{3/2}$ guarantees that X elements must be pushed from the smaller level X before the next recursive push occurs.

We want to get rid of the $X^{1/2}$ term, and therefore consider three cases:

1. $X \ge B^2$

In this case $X^{1/2}$ is dominated by sort(X), which can be shown by solving: $X^{1/2} \leq O(\frac{X}{B} \log_{M/B} \frac{X}{B}).$

2. $B \leq X \leq B^2$

In this case the $X^{1/2}$ term might be dominating. We now show that it is possible to find *one* block replacement that removes this term. The optimal replacement policy assumed in the ideal-cache model now guarantees to find a block replacement at least as good (see Section 2.3.2 on page 19).

The cause of the $X^{1/2}$ term was that we had to access one memory block in all down buffers on level $X^{3/2}$. Recall that a memory block is of size B. In some cases, however, a push does not transfer B elements to each down buffer. Consequently, we place the partially filled memory block from each of the $X^{1/2}$ down buffers in cache, and only transfer them to memory when they are filled with elements.

We now make use of the tall-cache assumption (see Section 2.3.1 on page 18) which states: $M = \Omega(B^2)$. As $X^{1/2} \leq B$ and because there only exists a constant number of levels covered by this case, all these blocks can be stored in a fraction of the memory, by choosing the constant in the tall-cache assumption, to the number of levels that are covered by this case.

Similarly, push also needs to access the $X^{1/2}$ pivot elements, while distributing the elements. Following the line of argument from above, it is also possible to hold the pivot elements of all levels covered by this case in memory at all times. This will eliminate the $X^{1/2}$ I/Os needed to read the pivot elements. This either doubles the constant in the tall-cache assumption, or requires that the pivot elements are always stored in the last block of the down buffer—the one kept in memory.

As it is possible to store the partially filled down buffers in memory, and only write full memory blocks to disk, as well as read the pivot element from these memory blocks, the $X^{1/2}$ term can be removed. Thus, the amortized cost of a push on level X covered by this case is sort(X).

3. $X \leq B$

The levels covered by this case have size less than $B^{3/2}$, so by the tall-cache assumption, these levels can all be stored in the memory, and therefore does not have an I/O cost.

The amortized I/O cost of a push on level X in all three cases is now shown to be sort(X). As shown, this can be amortized between the X elements, so that the amortized cost per element is:

$$O\left(\frac{1}{B}\log_{M/B}\left(\frac{N}{B}\right)\right).$$

The total amortized I/O cost of inserts and extracts on all levels can now be computed by summing the cost of the push and pulls performed on each level. This sum is:

$$\sum_{i=0}^{O(\log \log N)} O\left(\frac{1}{B} \log_{M/B}\left(\frac{N^{(2/3)^i}}{B}\right)\right),$$

which is dominated by the cost incurred by the largest level N. All in all the amortized I/O complexity of an insert or an extract in a Distribution Heap is:

$$O\left(\frac{1}{B}\log_{M/B}\left(\frac{N}{B}\right)\right).$$

This matches the optimal I/O bound achievable for priority queues.

Work Complexity of Insert and Extract

The work complexity of insert and extract was not described in the article by Arge et al. [6]. We now describe how it can be computed in much the same way as the I/O complexity: A push of X elements from level X to level $X^{3/2}$ sorts the X elements from the up buffer on level X and distribute them to the $X^{1/2}$ down buffers on level $X^{3/2}$. This step can be performed in $O(X \log X + X^{1/2}) = O(X \log X)$ work, which is $O(\log X)$ amortized.

Furthermore, a pull of X elements from level $X^{3/2}$ incurs $O(X \log X)$ work, in order to sort and distribute the three first down buffers. If level $X^{3/2}$ does not contain enough elements, the $X^{3/2}$ elements recursively pulled from level $X^{9/4}$ are first sorted and distributed in $O(X^{3/2} \log X^{3/2})$ work, which can be amortized over the $X^{3/2}$ elements to incur the amortized work per element of $O(\log X^{3/2}) = O(\log X)$.

As the amortized work complexity of a pull and a push is $O(\log X)$, the total cost of all push and pulls that occur during N/2 operations are:

$$\sum_{i=0}^{O(\log \log N)} O\left(\log N^{(2/3)^i}\right).$$

Recall that the largest level is dominating, and this sum is thus dominated by the work charged to level N.

The amortized work complexity of insert and extracting elements to/from a Distribution Heap is thus $O(\log N)$, which corresponds to the optimal work complexity for priority queues.

3.1.4 Space Complexity

The space consumption of the Distribution Heap, can be computed by first looking at the space consumption when the structure has just been rebuild. The total space consumption of a Distribution Heap, where the largest level has size N is:

$$3\sum_{i=0}^{\log_{3/2}\log_c} N^{(2/3)^i}.$$

This sum is naturally dominated by the term from the largest level, and the space consumption after a global rebuild is thus O(N). As the Distribution Heap is rebuild after N/2 operations the number of elements contained in the Distribution Heap will be in the range between N/2 and 3N/2. This ensures that the space consumption of the Distribution Heap at all times will be O(N).

3.1.5 Delete

Arge et al. briefly mentioned an extension to the Distribution Heap to enable it to support the deletion of elements as well. This is done by using propositions from [4, 32]. The technique is based on inserting *delete-elements*. A delete-element with priority x is placed in the data structure in the same way as an ordinary element with the same priority. On its way up through the data structure it will at some point meet an ordinary element with the same priority. When they meet, the ordinary element is deleted. This way, the delete is carried out in a lazy fashion which ensures that the I/O complexity of a delete is identical with that of an insert or an extract.

There are however several practical limitations, not described by Arge et al., which influence a implementation of a Distribution Heap which offers a deletion of elements:

First, the algorithms we use as competitors to the cache-oblivious priority queues do not offer delete, so we are not able to compare the performance of delete with other priority queues.

Second, because of the lazy fashion in which the data structure moves elements between levels, it is not clearly defined how a delete operation should work. There are two possibilities:

- 1. An element with the specified value is deleted. This implies that the deleteelement lives until it meets the first ordinary element in which case both elements are removed.
- 2. All elements with the specified value are deleted. A delete-element must, in this case, survive each encounter with ordinary elements, in order to eventually delete them all.

Deciding which solution is appropriate, may depend on the application for which it should be used for. It is therefore difficult to make a decision, without this knowledge.

Third, to implement the delete-element we need some way to distinguish ordinary elements and delete-elements. Implementing this with STL interface, where the elements can have any type, one extra bit must be added to all elements.

Furthermore, the expected behavior would be that a delete-element only deletes elements that were in the data structure *before* the delete-element itself was inserted. This implies that all elements have to be time stamped which requires

extra work and extra space—for example 31 bits which together with the "type"bit above make up for a word-aligned type.

As an example, adding extra 4 bytes for type and time stamping, will double the space-usage of each of the elements stored in a priority queue of integers, and thus also the total space usage. If the data structure is used with integers, this implies that the space requirements, when supporting delete, are twice as big as when not supporting this operation. This would result in an unfair comparison with other priority queues, which do not support delete.

Finally, there is also an issue with how long a delete-element should live: There is no way for a delete-element to determine when it is "done", and therefore it could potentially live in the data structure forever. However, during rebuilding delete-elements that have done their duty can be removed, because here all elements are sorted and it would therefore be possible to clean up.

Because the observations above make it difficult to make the right design decision and perform fair benchmarks, we have chosen not to include delete in our implementation and performance comparison.

3.1.6 Global Rebuilding

There are two reasons why the Distribution Heap has to be rebuild occasionally. First, the rebuilding is used in the analysis of the complexity of insert and extract in Section 3.1.3 on page 36. Second, the structure is designed to have a static size, where all buffers are allocated sequentially in memory. To create a dynamic data structure the global rebuilding is used to remove the old static structure, and create a new structure, where the largest level has size N.

To perform a global rebuilding operation, all elements are removed from the Distribution Heap. A new Distribution Heap is now created and the elements are inserted into this. The new Distribution Heap is created such that the sizes are computed based on the current number of elements in it. This implies that the size dynamically adapts to the usage of it.

When filling the elements into the new Distribution Heap, it is filled so that all down buffers on level X except the largest level N are filled with $X^{2/3}$ elements. This corresponds to the situation where all down buffers are half full. At the same time all up buffers are left empty. The remaining elements are filled into $\Theta(N^{1/3})$ down buffers on level N.

The global rebuilding ensures that X elements will have to be pushed to, or pulled from, level X before a recursive push to level $X^{3/2}$ is necessary. Specifically, on the largest level N this ensures that N elements must be pushed to level N, before a push to the next non-existing level $N^{3/2}$ would be needed. As the structure is rebuild after N/2 operations, the new structure will have a size that ensures that a recursive push would never have to access non-existing levels.

When the elements are reinserted in the priority queue, they need to be sorted in order to find the elements that go into each down buffer. One approach is simply to move all elements into a temporary structure and sort them before reinserting them. This has to use a cache-oblivious optimal sorting method to ensure an optimal cache-oblivious bound of sort(N) for the rebuilding of the Distribution Heap. Another approach is to exploit the fact that the elements are already stored in a priority queue so they can easily be extracted. This naturally ensures that the elements are sorted. This yields a bound of

$$N * O\left(\frac{1}{B}\log_{M/B}\left(\frac{N}{B}\right)\right) = sort(N).$$

We implement both types of rebuilding in order to investigate which one is best in practice.

3.1.7 The Constant *c*

The recursive definition of levels in the Distribution Heap stops once the level size becomes smaller than a constant size c. This is therefore a parameter that can be used for tuning the behavior of the Distribution Heap. The algorithms described above do, however, impose some constraints on the minimal allowed value of c, which are not described by Arge et al.

The computations of sizes of buffers and levels in a Distribution Heap will in many cases not be integers. It does not, however, make sense to have buffers of non-integer sizes. To ensure that there is enough space in each buffer, we simply round up to the nearest integer.

An important constraint is that, for a pull to work successfully, all levels must contain at least three buffers. This is because a pull needs elements from up to three down buffers, in order to guarantee that there are enough elements to pull. As we use a pull on level c to fill the extraction buffer, we need to be able to perform pull on level c. The size of c must therefore be chosen, so that even the smallest level contain at least three down buffers. This puts the following requirement on c, where $c \in \mathbb{N}$:

$$\left\lceil c^{1/3} \right\rceil \ge 3 \iff c^{1/3} > 2 \iff c > 2^3 \iff c > 8.$$

This implies that c is allowed to be any value larger than 8.

To choose an appropriate value for use in our benchmarks, we experimented with different values of c. The experiments showed that some of the values larger than 9 resulted in a slightly better performance, and that the higher the value the better performance. Which values that results in the improved performance is however dependent on both the size of the data set and on the sequence of operations which is performed. We therefore chose to use the value 9 in all our benchmarks.

3.1.8 Elements with Identical Priorities

The invariants described in Section 3.1 do not take into account the possibility that more than one element in the priority queue can have identical priorities. In

real-life applications this property is desirable, as some algorithms depend on this possibility, e.g., the Minimal Spanning Forest algorithm [6]. We therefore extend the functionality of the Distribution Heap to this case too.

To do this, we start by changing the invariants so that all relations are changed to be *less-than-or-equal*. The altered invariants are:

1. At any level, the elements are sorted *among* the down buffers, so that each element in a down buffer are smaller than, **or equal to**, any element in the next down buffer. That is, for any down buffer *a* on level *X* (called d_a^X) where $a \in \{1, 2, ..., X^{1/3} - 1\}$ any element *f* in d_a^X is smaller than any element *g* in d_{a+1}^X :

$$\forall f \in d_a^X, \quad \forall g \in d_{a+1}^X : f \leq g.$$

2. Any element f, in a down buffer on level X is smaller than, or equal to, any element g in the up buffer u^X on the same level. That is, for any $a \in \{1, 2, ..., X^{1/3}\}$:

$$\forall f \in d_a^X, \quad \forall g \in u^X : f \leq g.$$

3. Any element f, in a down buffer on level X is smaller than, or equal to, any element g in a down buffer in the level above $(X^{3/2})$. That is, for $a \in \{1, 2, ..., X^{1/3}\}$ and $b \in \{1, 2, ..., X^{1/2}\}$:

$$\forall f \in d_a^X, \ \forall g \in d_b^{X^{3/2}} : f \leq g.$$

These changes make sure that more than one buffer is allowed to hold elements with the same priority, and the invariants guarantee that all elements with the same priority are placed in consecutive buffers, so that the push and pull operations still achieve the expected result.

This does, however, make the implementation of the splitting of a full down buffer more complicated. The implementation needs to handle the case where many of the elements in the buffer have the same priority. A special case is when the median in the buffer has the same priority as the pivot element. In this case, it is not possible to ensure that the elements are equally distributed between the two buffers simply by comparing the elements with the partition element. If the elements are not equally distributed, the invariant that all buffers are at least $\frac{1}{4}$ full might be violated. This is handled by ensuring that half of the elements are placed in each buffer. Figure 3.4 illustrates how a split is handled in this special case.

3.1.9 Finding the Median Cache Obliviously

A frequently used operation in the Distribution Heap is splitting a buffer. For this, a suitable operation to find the median element is necessary. This is used to partition

Buffer before the split:	11, 11, 11, 9, 11, 11, 11, 11, 8, 11

Pivot element is: 11, the median is: 11.

If this special case is not handled, the result would be:

The first down buffer:	11, 11, 11, 11, 11, 11
The second:	9.8

The handling of the special case makes sure to balance the buffers:

11, 11

The first buffer:	11, 11, 11, 11, 11
The second:	11, 11, 9, 8, 11

Figure 3.4: A split of a buffer where the pivot element and the median element have the same priority.

the buffer into two equally sized buffers. One of the standard text-book algorithms for selecting a k'th element in worst case linear time [19, p. 189], is proven by Frigo et al. [22] to have the cache-oblivious I/O bound of scan(N). In the following we will call this method select.

This algorithm is, however, fairly slow, as the constants in the work complexity are fairly high. To improve this, Yde [56] has made an implementation called lazy_select, that performs better on average. To be able to use this optimized version, we will prove that this method also has a worst-case behavior that is cache obliviously optimal.

The basic idea of lazy_select, is that a random sample of the elements are drawn, and this sample is partitioned into three parts. Where the middle of these parts is chosen to contain a fairly small amount of the sample. The preferred case is now that the k'th element is found in the small middle section. If this is the case, the element is found in this section by using the ordinary version of select. The size of the sample, and the method for randomly selecting the elements for the sample, is designed to maximize the probability that this case will occur. However, if this fails, lazy_select defaults to the ordinary select algorithm. For further description see [56].

We now investigate the I/O complexity of lazy_select. The first part is that the sample is selected. This is done by randomly selecting

$$\log^{1/3}(N) \left(rac{N}{R}
ight)^{2/3}$$

elements from the input as described by [47]. Where R should be chosen to the block size of the cache. Yde's [56] program hard-codes R to be 16, which is there-

fore used below. This is used to convert an originally cache-aware method into an cache-oblivious implementation, because this is a small constant not chosen based on memory sizes. We now want to show that this I/O complexity is dominated by scan(N):

$$\log^{1/3}(N) \left(\frac{N}{16}\right)^{2/3} \leq scan(n) \iff$$

$$\log^{1/3}(N) \left(\frac{N}{16}\right)^{2/3} \leq \alpha \frac{N}{B} \iff$$

$$\log^{1/3}(N) \leq \alpha \frac{N}{B} \left(\frac{N}{16}\right)^{-2/3} \iff$$

$$\log^{1/3}(N) \leq \frac{\alpha}{B} \left(\frac{1}{16}\right)^{-2/3} N^{1/3} \iff$$

$$\log(N) \leq \left(\frac{\alpha}{B}\right)^3 \left(\frac{1}{16}\right)^{-2} N \iff$$

$$\frac{\log(N)}{N} \leq \left(\frac{\alpha}{B}\right)^3 \left(\frac{1}{16}\right)^{-2}$$

As the left-hand-side is always less than one, this can be made true by selecting α so that the right hand side is always larger than one. This is the case if

$$\alpha \ge \frac{B}{16^{2/3}},$$

in which case the number of I/Os incurred by the reading of the sample is dominated by scan(N).

Based on scanning the sample, the elements for partitioning the input into the three sections are now selected using scan(S), where S is the size of the sample. Then the input is partitioned into the three sections, using scan(N) I/Os. To find the element, the standard selection method is used on the section containing the element in scan(T), where T is the size of the section. As both S < N and T < N the total I/O complexity is scan(N).

3.1.10 STL Interface

One seemingly small aspect to keep in mind, is that both cache-oblivious priority queues are described to be minimal ordered heaps, where the smallest elements are extracted. However, STL, as defined in the C++ standard [27], requires that extracting an element for priority queues, per default, returns the largest element. Changing the *ordering* of the priority queues might seem trivial. The standard has, however, another requirement that makes it a bit tedious: A priority queue has a parameter that specifies a strict weak ordering [48] to be used, when comparing

elements. This is by default the STL function less(). This implies that the comparisons needed in the implementation, is the opposite of the descriptions in the cache-oblivious priority queues or any other priority queue we have found. In other words, if an element is extracted from an STL heap, it will be the largest element in the heap, whereas extracting an element from the described Funnel or Distribution Heap will return the smallest element. We therefore have to implement a *maximum ordered* heap using a "less than" comparison function. This will be reflected in the pseudo code we will present for both priority queues.

3.1.11 Pseudo Code

In the original article by Arge et al. [6] the algorithm is only explained in text. Furthermore, there are a couple of small details which need to be considered when actually implementing the algorithms. In order to make an unambiguous presentation, we therefore include pseudo code for the Distribution Heap. This pseudo code includes all the adaptations and changes we have made in this chapter, and therefore reflects our implementations.

Algorithms 1 to 5 show pseudo code for the Distribution Heap.

Algorithm 1 Distribution Heap: insert Input: element x 1: insert x into insert buffer I 2: sort(I) 3: if I is full

- 4: $\alpha \leftarrow I$ and extract buffer *E* merged together
- 5: push($d^{2/3}$ smallest elements from α) into level 1[†]
- 6: $I \leftarrow \{\}$
- 7: $E \leftarrow$ remaining part of α

[†]As the smallest level is of size d, $d^{2/3}$ is the number of elements that must be pushed into the first level to ensure the optimal complexity.

Algorithm 2 Distribution Heap: push to level X

Input: array A of size $X^{2/3}$ 1: sort(A)2: **if** there are no down buffers 3: first down buffer $B \leftarrow A$ place last element of A as pivot element in B4: 5: **else** $p \leftarrow \text{pivot of } B$ 6: while A contains elements 7: **if** first element in A < p8: if B is the last down buffer 9: insert remaining elements of A into up buffer[‡] 10: 11: else $B \leftarrow B + 1$ (proceed to next down buffer) 12: $p \leftarrow \text{pivot of } B$ 13: **if** |B| = 014: if $|A| \ge \frac{1}{2}X^{2/3}$ 15: $B \leftarrow A$ 16: place last element of A as pivot element in B17: else 18: insert remaining elements from A into the up buffer^{\ddagger} 19: else 20: if B is not full 21: insert first element of A into B22: else 23: if no free down buffers 24: empty last down buffer into up buffer[‡] 25: $e \leftarrow \text{first element of } A$ 26: $p_{next} \leftarrow \text{pivot of next down buffer}$ 27: if e = p and $e \leq p_{next}$ 28: $B \leftarrow B + 1$ (proceed to next down buffer) 29: $p \leftarrow p_{next}$ 30: else 31: $B_1, B_2 \leftarrow \operatorname{split}(B)$ 32: 33: set pivot elements for B_1, B_2

[‡]When the up buffer is full, the content is pushed to the next larger level before the remaining elements are inserted into the up buffer.

Algorithm 3 Distribution Heap: split

Input: down buffer *B* 1: $B_{new} \leftarrow$ new down buffer 2: pivot of $B_{new} \leftarrow$ pivot of B3: $B_{tmp} \leftarrow$ temporary buffer 4: $m \leftarrow \text{median of } B$ 5: pivot of $B \leftarrow m$ 6: for all elements x in B7: if x < mmove x to B8: 9: if x > m10: move x to B_{new} 11: if x = minsert x into B_{tmp} 12: 13: ensure equal size of B and B_{new} by distributing elements from B_{tmp}

Algorithm 4 Distribution Heap: extract

Output: element *x*

1: **if** |E| = 0

2: $E \leftarrow \text{pull from level 1}$

- 3: $x \leftarrow$ largest element from either insert or extract buffer
- 4: remove x

Algorithm 5 Distribution Heap: pull from level X

Output: buffer B of size $X^{2/3}$

- 1: $s \leftarrow$ number of elements in down buffers
- 2: if $s < \frac{3}{2}X^{2/3}$
- 3: sort(up buffer)
- 4: **if** level above exist
- 5: Y \leftarrow pull from level $X^{3/2}$
- 6: $u \leftarrow$ number of elements in up buffer
- 7: $Z \leftarrow merge Y$ with up buffer
- 8: insert u largest elements of Z into up buffer
- 9: insert remaining elements into $X^{1/2}$ down buffers
- 10: **else**
- 11: insert up buffer into down buffers
- 12: S \leftarrow sort first three down buffers
- 13: move $X^{2/3}$ elements from *S* into *B*
- 14: place remaining elements of S in 1, 2 or 3 down buffers
- 15: return B

3.2 The Funnel Heap

Compared with the Distribution Heap, the Funnel Heap is based on the opposite approach—here streams of elements are *merged*. The basic structure of a Funnel Heap consists of funnels, buffers and binary mergers [14]. We start by giving a short description of a funnel, and continue with the Funnel Heap after that.

The concept of a funnel was described by Prokop [44], who used it in a cacheoblivious version of Merge Sort called Funnel Sort. A funnel can be seen as a tree-like construction where the nodes in the tree are mergers that merges two input streams. The input streams are in buffers that are placed on the edges of a tree. The funnel is laid out in memory using the van Emde Boas layout described in Section 2.4. By choosing the size of the buffers in this tree to certain values, Prokop showed that a funnel gives optimal sorting in the cache-oblivious model. Figure 3.5 shows the structure of a funnel.



Figure 3.5: The structure of a 16-way funnel. Shaded regions are the occupied parts of buffers. (Reproduced from Brodal & Fagerberg [14] with permission from the Authors.)

To sort a number of elements with Funnel Sort, the funnel merges sorted input sequences into it's output buffer. The input buffers are sorted by recursively using Funnel Sort.

The Funnel Heap utilizes the funnel to construct a priority queue. The intuitive idea of the Funnel Heap, is to construct a tree, similar to the tree used to construct a funnel. This tree is constructed of a number of funnels and buffers joined by mergers. The Funnel Heap works as a priority queue, by ensuring that the tree is minimum orders, so that the smallest elements are placed close to the top of the tree.

Brodal & Fagerberg define the Funnel Heap by using the recursively defined constants k_i and s_i to determine the sizes of the components. They are defined as follows:

$$egin{array}{rcl} (k_1,s_1) &=& (2,8), \ s_i &=& s_{i-1}(k_i+1), \ ext{and} \ k_i &=& \lceil \lceil s_i^{1/3} \rceil \rceil, \end{array}$$

where [[x]] is "x rounded to the nearest power of 2"³.

A Funnel Heap is made up of an insertion buffer and a number of links. The insertion buffer I has size s_1 , which by the base case is defined to be eight. A link i consists of:

- *k_i* buffers, called *S_{i,j}* for *j* ∈ {1, 2... *k_i*}. Each of these contains a sorted sequence of up to *s_i* elements.
- a k_i -way funnel (K_i) that merges the content of the $S_{i,j}$ buffers.
- a buffer B_i that is the output buffer for K_i . The size of the buffer is k_i^3 .
- a binary merger v_i . It merges elements from B_i , with elements from the link above, by extracting elements from the buffer A_{i+1} .
- a buffer A_i that is the output buffer for the binary merger v_i . The size of A_i is k_i^3 .

This structure is illustrated in Figure 3.6. The buffers in the Funnel Heap is laid out in memory in a way, which minimizes the number of I/Os during the operations of the data structure. This memory layout is illustrated in Figure 3.7. An example showing how a Funnel heap might look, is found in Figure 3.8.

Because a funnel is a tree of binary mergers connected through buffers, the Funnel Heap can be seen as one large binary merger tree with buffers connecting these mergers. In the Funnel Heap the invariant is that the elements are *minimum ordered* so that, when traversing any path from the root to a leaf, the elements will be encountered in sorted order—in other words, smaller elements are placed closer to the root.

³This can be found by computing $2^{\lceil \log_2 x \rceil}$.



Figure 3.6: Illustration of the structure of a Funnel Heap [14].



Figure 3.7: Illustration of the memory layout of a Funnel Heap.



Figure 3.8: Example illustrating a Funnel Heap.

It is easy to extract the smallest element from a Funnel Heap: If the top buffer, A_1 , is empty, it is filled by recursively invoking the binary mergers below. The minimum ordering of the tree guarantees that the smallest element is either the first element in the buffer A_1 , or any of the elements in the insertion buffer I. Extract is illustrated with pseudo code in Algorithm 6.

Algorithm 6 Funnel Heap: extract		
Output: element x		
1: if $ A_1 = 0$		
2: invoke merger v_1 to fill A_1 recursively		
3: $e_1 \leftarrow$ largest element into insert buffer		
4: $e_2 \leftarrow \text{first element in } A_1$		
5: $x \leftarrow \text{largest of } e_1 \text{ or } e_2$		

The insertion of elements is more complicated. First, a new element is inserted in the insertion buffer I. If the insertion buffer becomes full, the content is inserted in the priority queue by performing an operation called a *sweep*. Pseudo Code for insert is shown in Algorithm 7.

Alg	Algorithm 7 Funnel Heap: insert		
Inp	ut: element x		
1:	insert x into insert buffer i		
2:	if <i>i</i> is full		
3:	sweep()		

The sweep operation start by finding the left-most empty leaf buffer $(S_{x,y})$, where x is the number of the link, and y is the number of the empty leaf buffer. The elements on the path from A_x down to $S_{x,y}$ are removed to form a sorted stream σ_1 . All elements in all smaller links—links with numbers less than x—are extracted in sorted order to another sorted stream σ_2 , and using a binary merger these two streams are merged to form the stream σ . Now, σ is used to refill buffers in the structure. First the buffers on the path from the root (A_1) to the empty buffer $(S_{x,y})$ are filled with the same number of elements as they contained before the sweep. This procedure guarantees that the minimum ordering of the elements is maintained.

An important observation is that a sweep to a leaf buffer in link x, empties all funnels and leaf buffers in all links i, where i < x. A sweep moving elements to link x, therefore ensures that a number of sweeps will have to be made before a new sweep access elements in link x. As a sweep empties all leaf-buffers in the links i, where i < x, and each new sweep to link i uses one extra leaf buffer in that link:

$$\prod_{a=0}^{x-1} s_a$$

sweeps will have to be performed before another sweep will need to move elements to link x.



Figure 3.9: Illustration of a *sweep* in the Funnel Heap.

Figure 3.9 is an example of a sweep in a Funnel Heap with 3 links. The first empty S-buffer is in this case $S_{3,2}$, which is marked by a star in the figure. The sweep starts by locating this buffer. The elements in buffers in the area marked σ_1 are removed to form the sorted sequence σ_1 . The elements in the buffers in the area marked σ_2 are then extracted by removing the connection between v_2 and A_3 , and repeatedly calling extract on the remaining merger tree. Then, σ_1 and σ_2 are merged to form σ , and the connection between v_2 and A_3 is reestablished. The elements in σ are now inserted in the priority queue again.

The buffers on the path from the root of the merger tree, down to the empty S-buffer, are refilled with elements from σ . Finally, the remaining elements are moved to $S_{3,2}$. These buffers are all shaded in the figure.

Algorithm 8 shows pseudo code for the sweep operation.

3.2.1 Complexity of Insert and Extract

For the Distribution Heap it was necessary to discuss the analysis from the original article, because we use it in Chapter 4.

As we do not use the analysis of the Funnel Heap again we therefore only summarize the results from the article by Brodal & Fagerberg here.

Algorithm 8 Funnel Heap: sweep

- 1: $i \leftarrow$ first link with an empty S buffer S
- 2: record sizes of buffers A_1 to A_i and B_i
- 3: record sizes of buffers on path P in k_i
- 4: create streams σ_1 , σ_2 and σ
- 5: insert elements from A_i into σ_1
- 6: insert elements from B_i into σ_1
- 7: insert elements on path P into σ_1
- 8: $\sigma_2 \leftarrow$ extract all elements in link 1 to i 1
- 9: $\sigma \leftarrow \text{merge } \sigma_1 \text{ with } \sigma_2$
- 10: refill buffers $A_1 A_i$ and B_i to recorded size with elements from σ
- 11: refill buffers on path P to recorded size with elements from σ
- 12: fill S with remaining elements from σ

In Brodal & Fagerberg [14], it is shown that the amortized I/O complexity of an insert operation is

$$O\left(\frac{1}{B}\log_{M/B}\left(\frac{N}{B}\right)\right).$$

The data structure is analyzed such that all I/Os are charged to the insert operation, so that the amortized I/O complexity of an extract is zero. With these bounds the Funnel Heap is therefore optimal in I/O like the Distribution Heap.

The work complexity of the Funnel Heap is not described in [14], but it can be analyzed in the same way as the I/O complexity. The amortized work complexity of one insert or extract operation can therefore be shown to be $O(\log N)$ which is the same bound as the Distribution Heap and optimal for priority queues.

The space complexity is not analyzed in [14] either, even though the data structure appears to have exponential growth which could be a major practical problem. We therefore take a close look at the space usage of the funnel Heap in the following two sections, where we analyze and improve the space usage.

3.2.2 Dynamic Structure

As it was the case with the Distribution Heap, the Funnel Heap is a static structure. In Brodal & Fagerberg [14] all buffers are required to be placed consecutively in memory. This is not feasible in a real implementation as we would like a dynamic structure that adapts to the use and automatically grows when needed. To insure this we have chosen to alter the allocation method slightly, so that each link can be allocated individually. When a Funnel Heap is created we therefore only allocate space for the first link, and if/when it exceeds the current capacity, the next link is allocated. This ensures a dynamic structure that automatically grows when needed. The disadvantage is that the described requirement, that all buffers must be placed in consecutive memory, cannot be met, as the memory system might allocate the new space anywhere. The possible loss by this approach is, however, only that the last S-buffer in link i - 1 is not placed in front of the buffer A_i , and this will not have a noticeable effect on the number of I/Os, or the asymptotic running time.

The alternative to this method is to perform some sort of global rebuilding procedure whenever another link is needed. The global rebuild would extract all elements from the old Funnel Heap and construct a new Funnel Heap with the elements. This Funnel Heap would then be constructed with one additional link such that the capacity would be increased. Global rebuilding, however, require N extracts as well as a construction method, and we therefore expect the first method to be faster in practice.

3.2.3 Space Complexity

Another problem that occurs is that the space consumption of a link is too large for the described method to have a linear space complexity. A solution is to relax the requirements that everything has to be allocated sequentially, so that each of the leaf buffers could be allocated on demand. This implies that the first time a sweep needs a link, it creates the buffers A_i and B_i , the funnel K_i and the one leaf buffer needed (S_{i1}) . The rest of the leaf buffers is first allocated when another sweep need to move elements into it. This potentially increases the number of I/Os, but will not affect the asymptotic I/O complexity⁴. We now look at the complexity of this approach. The actual space complexity of the Funnel Heap is not treated in the original article, so we will here make a short analysis.

To find the actual space complexity of the Funnel Heap, we first recall that in a link *i* there are k_i S buffers of size s_i . By the definition of s_i and k_i we know that:

$$k_i = O(s_i^{1/3}).$$

The space consumption of the S buffers are thus:

$$k_i s_i = O(k_i k_i^3) = O(k_i^4)$$

Because K_i , A_i and B_i all are of size $O(k_i)$, the space used by the S buffers dominates the space used for a link *i*. Moreover, as s_i and k_i are increasing, the space consumption of a funnel heap with *i* links is dominated by the *i*th link. Brodal & Fagerberg [14] calculates the growth of *k* to be roughly $k_{i+1} = k_i^{4/3}$.

When a S buffers of size s_i is filled during a sweep, there is added $O(s_i)$ elements to it. When using the proposed solution to ensure linear space, the dominating buffer is thus always filled to a fraction of its capacity. This implies that while inserting elements the space complexity of the Funnel Heap will be linear. When elements are extracted from the Funnel Heap, the elements are not necessarily removed from the largest S buffers. To ensure linear space of shrinking Funnel Heap it would be necessary to add a *reverse sweep*, that moves elements from the largest

⁴This was not described in the article by Brodal & Fagerberg, but after some e-mail correspondence, it was revealed that this was in fact the intention of the authors.

link, to other links, when the Funnel Heap becomes to space. We have not devised such an operation.

The total space consumption of a Funnel Heap is thus linear in the largest number of elements that it has contained.

3.3 Construction of Priority Queues

In some applications of priority queues, it is possible to construct the priority queue with an initial set of elements. Sometimes, it is possible to offer a method that inserts this initial set of elements faster than if they where inserted one by one. When this occurs, a construction method is added, in order to make this performance gain available.

In the standard RAM priority-queue, it is possible to construct a priority queue with the work complexity O(N), instead of N inserts with the total work complexity of $O(N \log N)$ [19]. This leads to a lower bound in the cache oblivious model of scan(N) I/Os, which will be a great improvement compared to N inserts, that in total require sort(N) I/Os. At the same time, the process of inserting a large number of elements requires many element moves. Even if we are not able to find an asymptotically optimal method for constructing the priority queue, we expect to be able to improve the performance if we can find a simple method for the construction that does not require a large number of moves.

To construct a priority queue efficiently, we have to place the elements in the buffers so that the invariants are maintained in the priority queue. If this is done, we are then able to use the priority queue as otherwise described when extracting and inserting elements.

In the following we investigate whether it is possible to add a construction method that improves the performance of the two cache-oblivious priority queues.

3.3.1 Distribution Heap

The invariants in the Distribution Heap define how the elements are distributed between the buffers. This is done so that there is an ordering between the buffers but not between the elements inside a buffer. To achieve such a distribution of the elements, when constructing a Distribution Heap, the elements must be partitioned into intervals that fit into the different buffers.

Such a construction of a Distribution Heap is very similar to the global rebuilding, so we design the construct method to use the same approach (see Section 3.1.6 on page 40).

The method for rebuilding described in [6], requires that the input is sorted before is can be distributed in the down buffers in the Distribution Heap. This make the complexity of this construction method sort(N), which does not offer any asymptotic improvement over the repeated calls of insert. There might, however, be an improvement in the constants, as the construct method does not have to copy the elements that many times. This is most noticeable when N is large. Pseudo Code for the Distribution Heap construction is shown in Algorithm 9.

Algorithm 9 Distribution Heap: construction

Input: array A		
1: 8	$\operatorname{sort}(A)$	
2: 0	create the levels, by computing sizes top-down	
3:	$L \leftarrow \text{first level}$	
4: •	while $ A > 0$	
5:	for all down buffers B in L	
6:	insert $X^{1/3}$ elements into B from A	
7:	pivot of $B \leftarrow$ largest element in B	
8:	$L \leftarrow \text{next level above}$	

3.3.2 Funnel Heap

The invariant in the Funnel Heap is that all paths from the root to a leaf contains elements in sorted order. We present two procedures which place the elements in order to maintain the invariant.

In the the first method, all elements are sorted and then placed in the Funnel Heap in the following order: First, the smallest elements are inserted into A_1 and the next smallest elements are placed in B_1 . Then the funnel K_1 in link 1 is filled, so that the smallest elements are placed closest to the root of the funnel. Finally, the buffers $S_{1,x}$ for all $x \in \{1..s_i\}$ are filled. The remaining elements are recursively filled into the following links. This ensures that all paths from the root to a leaf in a Funnel Heap are sorted in increasing order. To perform this construct, it is required that all elements are sorted so the complexity is sort(N), which matches the complexity of N inserts. An important observation is that the buffers in a Funnel Heap are placed in a specific order in memory. This order is actually the same as the order we need to fill the buffers during the construction. Therefore, this method is simple to implement—simply sort the input and use the resulting array as the basic data storage for the Funnel Heap.

The other possibility is to fill only the leaf buffers in the structure $(S_{x,y})$. The advantage of this method is that the invariant does not put any restrictions on the relation between the buffers, so the only requirement is that each of the buffers is sorted internally. Therefore it is possible to divide the input into sections that fit into the leaf buffers, and then sort these sections. Because this method is only sorting smaller sections of the input, and not the complete input, the I/O complexity will be improved.

This construction method is based on sorting chunks for each of the needed S buffers in the Funnel Heap. The I/O complexity of this construction method can be

described by the expression:

$$\sum_{i=1}^{p} \sum_{j=0}^{k_i} sort(s_i),$$

where p is the number of links needed for the Funnel Heap to hold the elements. It is, however, important to note that all elements are not sorted at once—instead each chunk of size S_i is sorted separately during construction.

Note that it is cheaper to perform E sorts of F elements, than to sort EF elements:

$$E * O\left(\frac{F}{B}\log_{M/B}\left(\frac{F}{B}\right)\right) \le O\left(\frac{EF}{B}\log_{M/B}\left(\frac{EF}{B}\right)\right).$$

This shows that the entire sum shown above will be upper bounded by sort(N).

We have chosen to implement only the latter of the two methods, as it has, what looks like, the best behavior, and it is still fairly simple. Algorithm 10 shows pseudo code for the Funnel Heap construction.

Algorithm 10 Funnel Heap: construction

Input: array A 1: $i \leftarrow 1$ 2: while |A| > 03: create link i4: for all S buffers S in i5: $v \leftarrow |S|$ 6: $S \leftarrow$ sort first v elements from A 7: $i \leftarrow i + 1$

3.4 Sort Method

For the implementation of both algorithms, we need a sorting method. We have chosen to implement two versions, so that we are able to compare the performances. The first uses an implementation of Funnel Sort [44], implemented by us for the purpose. Because Funnel Sort is proven to be optimal in the cache-oblivious model, the versions that use this implementation are cache oblivious implementations with the optimal I/O complexity.

By studying the sizes of the buffers that need to be sorted, one notices that the size of these buffers grows at a slower rate than the size of the complete priority queue. In other words, the size of the buffers is only a fraction of N. Another observation is that standard RAM-model sorting algorithms often have an I/O complexity fairly close to the optimal sorting bound. It has e.g. been shown by Demaine in [20] that a standard 2-way merge-sort has the I/O complexity

$$\Theta\left(\frac{N}{B}\log_2\frac{N}{B}\right).$$

These observations give the impression that the optimal memory behavior of sorting might not be that important, so we have also made a version of the implementations that uses RAM-model sorting. This second version uses STL's standard sorting method [40, 48] to sort elements.

By comparing these two implementations, we will be able to determine how the two sorting methods perform when they are used for construction.

3.5 Limited Address Space

We now look into a couple of issues regarding the address space available on contemporary computers.

The most widespread computer platform today, is the Intel x86 architecture, which is used in most consumer PCs. This platform uses CPUs from Intel, like the Pentium series, but other companies, like AMD, are also supplying compatible CPUs (see Figure 1.3 on page 9 for a list of some contemporary CPUs). The x86 architecture uses 32-bits for pointers and can therefore only access 4Gb address space. Even though Intel has made an extension to their 32-bit CPUs, called *PAE* [26], which allows the kernel to indirectly (with a few percentages of overhead) address up to 64GB (2^{36}). This does not, however, influence the amount of available memory for a single process.

64-bit computers have emerged which can address in the order of exabytes (2^{60} bytes) , which is sufficient for current memory and storage technologies. 64-bit CPUs, like the Sun UltraSparc, Digital Alpha or HP PA-Risc, are available but none of these are as widely used as the x86 architecture. Recently, Intel released their 64-bit Itanium CPU and AMD is on their way with a 64-bit CPU called Hammer.

Another limitation is that some operating systems restrict the amount of data that can be allocated to the amount of actual available swap space. This is a safety feature in operating systems to prevent programs from crashing when running out of memory. This way, programs can handle out-of-memory situations without crashing. An example of this is Solaris where the total amount of memory that can be allocated, is limited to the amount of swap space. This is called *conservative allocation*. In Linux, it is possible to change between different policies. In one of them, there is no limit on the amount that can be allocated, which is called *over-committed allocation*. This is based on the observation that programs frequently allocate more memory than they actually use. An example could be a program that allocated a large array, and "in order to have it big enough" the programmer allocates it with a size of several MB, although the program only uses the first few KB. With an over-committed allocation scheme, the operating system will tell the program that the allocation of the whole array succeeded, but physical

memory will not be allocated until it is needed. This, of course, involves the risk that any program can run out of memory at any given point in time, but this is a choice of the user of the computer, which makes it possible to run more programs simultaneously.

Contrary to external-memory algorithms, which only work on a small part of data at the time, the cache-oblivious approach does not know anything about memory sizes and therefore has to allocate memory corresponding to the whole data set. When working with cache-oblivious data structures on 32-bit computers these issues limits the maximal data set that can be worked on. This puts a limit on the scalability of cache oblivious data structures.

As an example of this, we turn to the Distribution Heap: In the situation, when a global rebuilding is performed, all N elements are extracted to a temporary location, and a new Distribution Heap is build and filled with the elements. As the global rebuilding ensures that the new Distribution Heap contains elements in at most 1/3 of the space, this results in an address space consumption of at least 4N during a rebuild. This implies that a Distribution Heap can at most utilize 1/4 of the available memory space. This is, in fact, the worst case for the address space complexity, whereas the constant in the space complexity at times *can* be larger. This happens just before a rebuild procedure which would shrink the data structure. In this case the Distribution Heaps uses even more space per element. On a 32-bit computer with Linux, approximately 2GB is available for each process. The maximal amount of data that can be stored in a Distribution Heap is thus 500MB, which would not leave any room for the program code, and the usage of the data. For data sizes larger than this limit, the only solution is to use cache-aware algorithms that explicitly stores data on the disks.

If the full potential of the cache-oblivious data structures is to be used, the amount of memory that can be addressed should not be limited. This could be done by running on 64-bit computers with an over-commit allocation scheme. Even though, for example, Solaris and HP-UX are running on 64-bit processors, they do not allow over-committing. Linux, on the other hand, does exactly this, so running Linux on for example the Intel Itanium, AMD Hammer or a Digital Alpha CPU would be a viable approach.

Chapter 4

A Cache-Oblivious Priority Deque

Beware of bugs in the above code; I have only proved it correct, not tried it.

— Donald E. Knuth

In this section, we introduce a cache-oblivious priority deque by combining ideas from a RAM-model *priority deque* with those from the Distribution Heap. This algorithm is inspired by our work with the cache-oblivious priority queues.

A standard priority queue only supports extraction of either the smallest or the largest element. An interesting extension, is to allow for the extraction of both the smallest **and** the largest elements from the same data structure. This extended data structure is called a *priority deque*, where deque is an abbreviation of *Double Ended QUEue*. A priority deque supports the following operations:

- **Insert**(x) adds the element x to the data structure.
- **Extract_min()** returns the smallest element currently stored in the data structure and removes it.
- Extract_max() returns the largest element currently stored in the data structure and removes it.

Priority deques can be used for various applications: External Quicksort, constanttime Find Median, logarithmic-time Delete Median [8], as well as applications related to sorting and computational geometry problems [53].

We start by describing the approaches previously used to construct RAMmodel priority deques. Following this, we present our new cache-oblivious priority deque.

4.1 RAM-model Priority Deques

Different RAM-model algorithms for priority deques have previously been published in [8, 17, 30, 53]. The common approach for designing priority deques, is to start with a standard priority queue, and try to adapt this data structure in order to achieve the double-ended property. In this section, we give a summary of these ideas.

Twin Heap

In [30, p. 157], Knuth gives the exercise: Construct a priority deque by placing two priority queues "back-to-back". The solution suggests ordering the elements in twin-pairs. Each pair is then ordered so that the first element is smaller than the second element. A priority deque is then constructed, using two priority queues: One *minimum ordered* containing the small element of each pair, and one *maximum ordered* containing the large element of each pair. When elements are added and extracted, the pairs and priority queues are updated to maintain the invariants. Special handling must be done in the cases where an uneven number of elements exists.

The idea of maintaining two parts, one minimal ordered and one maximum ordered, is the general approach in all the other solutions too.

Min-Max Heap

In this data structure, proposed by Atkinson et al. [8], a normal heap is adapted so that odd levels are minimum ordered and even levels are maximum ordered. The root node contains the smallest element, and the largest element is found in one of the two nodes on the second level (see Figure 4.1). This structure can be seen as a minimum heap and a maximal heap intermixed. The operations to move elements up and down the tree are modified to move elements across two levels at a time.



Figure 4.1: The Min–Max priority deque works by having alternating levels of small and large elements (from [42]).

The Deap

Carlsson's [16] "double ended heap"—or Deap—follows the twin–pair idea very closely. The twin–pairs are handled by storing the smallest element from each pair on the left side of a binary tree, and the largest elements on the right side (see Figure 4.2). The invariants are that the left side is a minimum ordered heap and the right side is a maximum ordered heap. Also, any element in the minimum side, is smaller than its *corresponding element(s)* on the maximum side—similar to the Twin Heap approach. The definition of the corresponding element was not precise in Carlsson's original article, but we suggested a correction of this, in our previous project [42]. When elements are extracted, the pairs and the ordering of them are reestablished to maintain the invariants.



Figure 4.2: The Deap maintains a horizontal split of the elements; keeping small elements on the left and large elements on the right (from [42]).

Interval Heap

The Interval Heap was developed by van Leeuwen and Wood [53]. The basic idea is to construct a tree where each node contains an interval. In an ordinary heap, each node stores the smallest element in the subtree below it. In the Interval Heap, each node contains the smallest **and** largest element of the subtree. If a node contains the values a and b, all elements in all children of this node are in the range [a, b] (see Figure 4.3). In a situation where an Interval Heap contains an uneven number of elements, the last node contains only one element a and represents the interval [a, a]. Intuitively, this heap has a minimum ordered heap in the left side of all nodes, and a maximum ordered heap in the right side of all nodes and the operations for handling elements are quite similar to a standard heap.

Our previous benchmarking of implementations of the three priority deque algorithms in [42], revealed the Interval Heap to be the most effective, mainly due to its simplicity.



Figure 4.3: In the Interval Heap each node contains an interval in which all children node's intervals are included. All minimum elements are stored in the left side of each node and maximum elements are stored in the right (from [42]).

4.2 A Cache-Oblivious Priority Deque

Inspired by the structure of the Interval Heap, we have developed a modified Distribution Heap which works as a cache-oblivious priority deque. We call this data structure *DIPD* (an abbreviation of *Distribution Interval Priority Deque*).

Recall that a level in the Distribution Heap consists of a number of down buffers holding the smallest elements, and one up buffer holding larger elements on their way up to higher levels. The smallest element is located in the leftmost down buffer, in the smallest level.

The idea of the DIPD, is to double the amount of down buffers and assign half of them to hold small elements (*min–buffers*), while the other half hold large elements (*max–buffers*). Subsequently, the up buffer now holds the *middle* elements, which are contained in the interval between the minimum and maximum elements on that level. All elements in the up buffer, and in all levels above, are in the interval between the largest element in the min–buffers, and the smallest element in the max–buffers.

In the Distribution Heap, the smallest elements are located in the smallest level of the priority queue, and the largest elements are stored in the top of the priority queue—or they are temporarily stored in up buffers on their way to the largest level. In the DIPD the structure is split vertically, so that small elements are placed in one half of each level, and large elements are placed in the other half of each level. See Figure 4.4.



Figure 4.4: The DIPD stores the smallest elements in the min– buffers (1–7) and the largest elements in the max–buffers (8–14). The up buffers A and B contain elements in the interval between the min–buffers and the max–buffers.

Formally, the DIPD maintains six invariants (modified from the three invariants of the Distribution Heap, see Section 3.1.8 on page 41):

1a. At any level, the elements are sorted *among* the min-buffers, so that each element in a min-buffer is smaller than, or equal to, any element in the next min-buffer. That is, for any min-buffer *a* on level *X* (called d_a^X), for $a \in \{1, 2, \ldots, X^{1/3} - 1\}$ any element *f* in d_a^X is smaller than any element *g* in d_{a+1}^X :

$$\forall f \in d_a^X, \quad \forall g \in d_{a+1}^X : f \le g.$$

1b. At any level, the elements are sorted *among* the max-buffers, so that each element in a max-buffer is smaller than, or equal to, any element in the next max-buffer. That is, for any max-buffer a on level X (called e_a^X), for $a \in \{1, 2, \ldots, X^{1/3} - 1\}$ any element f in e_a^X is smaller than any element g in e_{a+1}^X :

$$\forall \ f \in e_a^X, \ \ \forall \ g \in e_{a+1}^X : f \leq g.$$

2a. Any element f, in a min-buffer on level X is smaller than, or equal to, any element g in the up buffer u^X on the same level. That is, for any $a \in \{1, 2, \ldots, X^{1/3}\}$:

$$\forall f \in d_a^X, \ \forall g \in u^X : f \leq g.$$

2b. Any element f, in a max-buffer on level X is larger than, or equal to, any element g in the up buffer u^X on the same level. That is, for any $a \in \{1, 2, \ldots, X^{1/3}\}$:

$$\forall f \in e_a^X, \ \forall g \in u^X : f \ge g_A$$



Figure 4.5: Illustration of the intervals in the priority deque. Numbers and letters refer to buffers from the example in Figure 4.4 on the page before.

3a. Any element f, in a min–buffer on level X is smaller than, or equal to, any element g in a min–buffer in the level above $(X^{3/2})$. That is, for $a \in \{1, 2, ..., X^{1/3}\}$ and $b \in \{1, 2, ..., X^{1/2}\}$:

$$\forall f \in d_a^X, \ \forall g \in d_b^{X^{3/2}} : f \le g.$$

3b. Any element f, in a max–buffer on level X is larger than, or equal to, any element g in a max–buffer in the level above $(X^{3/2})$. That is, for $a \in \{1, 2, \ldots, X^{1/3}\}$ and $b \in \{1, 2, \ldots, X^{1/2}\}$:

$$\forall f \in e_a^X \quad \forall g \in e_b^{X^{3/2}} : f \ge g.$$

The implications of this structure on the buffers in Figure 4.4, are illustrated in Figure 4.5.

Invariants **1a**, **1b**, **3a** and **3b** ensure that all min–buffers and max–buffers represent disjoint intervals, and that these buffers are ordered as illustrated in Figure 4.5. We therefore use the notion of an ordering of the buffers, so when we talk about one buffer being smaller than another, we actually talk about the way the intervals—represented by the buffers—are related.

In the following, we describe how the operations in the Distribution Heap can be adapted, so that the six invariants in the DIPD are maintained.

4.2.1 Push

When the up buffer on level X is full, it must be pushed into the next level $X^{3/2}$. We need to adapt push to distribute elements to both min-buffers and max-buffers. Following the strategy from the Distribution Heap, we first sort the elements in the up buffer, and then distribute the smallest elements to the min-buffers. This maintains Invariant **1a**.

Once the next element is larger than the pivot element of the largest min–buffer, we continue by distributing elements to the max–buffers. This can be done by
reversing the distribution. We scan the sorted elements in reverse order, and distribute them to the max–buffers, starting with the largest max–buffer. This ensures that Invariant **1b** is maintained.

When an element smaller than the pivot element of the smallest max–buffers is encountered, we know that the remaining elements are in the interval between the min–buffers and max–buffers, and we can insert the remaining elements in the up buffer, while maintaining Invariants **2a** and **2b**.

Full up buffers, and elements with the same priority, are handled in the same way as in the Distribution Heap. When full up buffers are pushed recursively, the elements in the up buffer, by Invariants **2a** and **2b**, are all in the interval between the min–buffers and the max–buffers. When these elements are inserted in the next larger level, this ensures that Invariants **3a** and **3b** are maintained.

Algorithm 11 and 12 shows pseudo code for push. The split operation used by push and distribute_max is identical to split in the Distribution Heap (shown in Algorithm 3 on page 47), and is therefore not repeated here. At the bottom of the pseudo-code in this chapter, we have included descriptions of the differences between the operation in the DIPD, and the corresponding operation in the Distribution Heap.

I/O Complexity of Push

As the DIPD is based on the Distribution Heap, and follows the operations from it very closely, the complexity analysis of the two are very similar. In Chapter 3, we analyzed the complexity of the Distribution Heap, and we, therefore, only describe the differences between the two algorithms here.

The distribution of the elements between the min-buffers in push follows the same strategy as push in the Distribution Heap. The complexity of this is therefore still $sort(X) + X^{1/2}$. The distribution of the elements to the max-buffers has the same complexity as the distribution in the Distribution Heap, as the only difference is that it scans the elements and the down buffers in the reverse order. The amortized I/O complexity of one pull of X elements from level $X^{3/2}$ is then $sort(X) + 2X^{1/2}$. Note that the difference, compared with the complexity in the Distribution Heap, is the factor 2 on $X^{1/2}$. This only requires that the constant the in tall-cache assumption is doubled, for all the blocks to fit into the memory.

As a push from level X empties the up buffer, X elements has to be added to the up buffer on that level before another push is needed, so identically to the Distribution Heap, the cost can be amortized over these X elements.

Algorithm 11 DIPD: push to level X

Inp	ut: array A of size $X^{2/3}$
1:	sort(A)
2:	if there are no min–buffers
3:	first min–buffer $B \leftarrow A$
4:	place last element of A as pivot element in B
5:	else
6:	$p \leftarrow \text{pivot of } B$
7:	while A contains elements
8:	if first element in $A > p$
9:	if B is the last min–buffer
10:	distribute_max(remaining elements from A)
11:	else
12:	$B \leftarrow B + 1$ (proceed to next min–buffer)
13:	$p \leftarrow \text{pivot of } B$
14:	$\mathbf{if}\left B ight =0$
15:	if $ A \geq rac{1}{2} X^{2/3}$
16:	$B \leftarrow A$
17:	place last element of A as pivot element in B
18:	else
19:	distribute_max(remaining elements from A)
20:	else
21:	if B is not full
22:	insert first element of A into B
23:	else
24:	if no free min–buffers
25:	empty last min–buffer into up buffer [‡]
26:	$e \leftarrow \text{first element of } A$
27:	$p_{next} \leftarrow \text{pivot of next min-buffer}$
28:	$\mathbf{if} \ e = p \ \mathbf{and} \ e \le p_{next}$
29:	$B \leftarrow B + 1$ (proceed to next min–buffer)
30:	$p \leftarrow p_{next}$
31:	else
32:	$B_1, B_2 \leftarrow \operatorname{split}(B)$
33:	set pivot elements for B_1, B_2
T1	and shares in much is in lines 10 and 10 where a new around

The only change in push is in lines 10 and 19, where a new operation distribute_max is called, and that the comparison in line 8 is reversed.

[‡]When the up buffer is full, the content is pushed to the next larger level, before the remaining elements are inserted into the up buffer.

Algorithm 12 DIPD: distribute_max to level X

Inp	ut: array A of size $\leq X^{2/3}$
1:	if there are no max–buffers
2:	first max–buffer $B \leftarrow A$
3:	place last element of A as pivot element in B
4:	else
5:	$p \leftarrow \text{pivot of } B$
6:	while A contains elements
7:	if first element in $A < p$
8:	if B is the last max–buffer
9:	insert remaining elements from A into up buffer [‡]
10:	else
11:	$B \leftarrow B + 1$ (proceed to next max–buffer)
12:	$p \leftarrow \text{pivot of } B$
13:	$\mathbf{if} B = 0$
14:	if $ A \geq rac{1}{2}X^{2/3}$
15:	$B \leftarrow A$
16:	place last element of A as pivot element in B
17:	else
18:	insert remaining elements from A into up buffer [‡]
19:	else
20:	if B is not full
21:	insert first element of A into B
22:	else
23:	if no free max–buffers
24:	empty last max–buffer into up buffer ¹
25:	$e \leftarrow \text{first element of } A$
26:	$p_{next} \leftarrow \text{pivot of next max-buffer}$
27:	if $e = p$ and $e \le p_{next}$
28:	$B \leftarrow B + 1$ (proceed to next max-buffer)
29:	$p \leftarrow p_{next}$
30:	else
31:	$B_1, B_2 \leftarrow \operatorname{split}(B)$
32:	set pivot elements for B_1, B_2
di	stribute_max distributes elements to the max-buffers and finall

distribute_max distributes elements to the max-buffers and finally moves the remaining elements to the up buffer. The only difference between distribute_max and push in the Distribution Heap, is that input size might be smaller than $X^{2/3}$.

4.2.2 Pull

As we need to be able to extract both the smallest and largest elements, we need two versions of pull. The pull_min works as pull in the Distribution Heap, and extracts the X smallest elements from Level $X^{3/2}$. The pull_max pulls the X largest elements.

The only difference between pull_min and pull in the Distribution Heap, is that when the min–buffer contains too few elements and both the levels above and the up buffer are empty, it will need to fetch elements from the max–buffers. This is done by finding the smallest max–buffer and simply moving the elements in it, to a min–buffer. This is repeated until there are enough elements for the pull_min operation to continue. This is possible due to the fact that when both the levels above, and the up buffer is empty, we know that the interval between the min–buffers and the max–buffers is empty too. This ensures that the invariants are maintained. The pseudo code for pull is shown in Algorithm 13.

The pull_max works in the same way as pull_min, the only difference being that it extracts elements from the max–buffers instead.

Algorithm 13 DIPD: pull_min from level X		
Output: buffer B of size $X^{2/3}$		
1: $s \leftarrow$ number of elements in min–buffers		
2: if $s < \frac{3}{2}X^{2/3}$		
3: if both up buffer and level above are empty		
4: move content of smallest max–buffer to an empty min–buffer		
5: else		
6: sort(up buffer)		
7: if level above exist		
8: $Y \leftarrow pull from level X^{3/2}$		
9: $u \leftarrow$ number of elements in up buffer		
10: $Z \leftarrow merge Y$ with up buffer		
11: insert u largest elements of Z into up buffer		
12: insert remaining elements into $X^{1/2}$ min–buffers		
13: else		
14: insert up buffer into min–buffers		
15: S \leftarrow sort first three min–buffers		
16: move $X^{2/3}$ elements from S into B		
17: place remaining elements of S in 1, 2 or 3 min–buffers		
18: return B		

The only change made is that the if-then statement in lines 3–4 handles the case when elements must be moved from a max–buffer to min–buffer.

I/O Complexity of Pull

The analysis of pull_min and pull_max are identical, so we only analyze pull_min. The majority of this algorithm is a repetition of pull in the Distribution Heap. This part has the complexity for pulling X elements from level $X^{3/2}$ of sort(X).

When the pull_min need to move the content of a max-buffer to the min-buffer, this is performed by scanning the max-buffer, and copying the elements to an empty min-buffer. All in all, this requires scan(Y) I/Os to move Y elements.

As Y = O(X), the I/O cost of pull in the DIPD is dominated by the I/O cost found for one pull in the Distribution Heap of sort(X).

4.2.3 Insert, Extract_min and Extract_max

Again, we need two functions to extract elements from the DIPD. We therefore make an extra extraction buffer, so that one buffers small elements extracted from the DIPD using pull_min, and the other buffers large elements extracted using pull_max. When elements are inserted and extracted, the three buffers are maintained in the same way as in the Distribution Heap.

Algorithm 14 and 15 shows pseudo code for insert and extract_min. The extract_max operation can be found by making trivial changes to extract_min, so the pseudo code is not shown here.

Algorithm 14 DIPD: insert
Input: element x
1: insert x into insert buffer I
2: $\operatorname{sort}(I)$
3: if <i>I</i> is full
4: $\alpha \leftarrow I$ and extract buffers E_{min} and E_{max} merged together
5: $e_{min} \leftarrow E_{min} $
6: $E_{min} \leftarrow \text{the } e_{min} \text{ smallest elements from } \alpha$
7: $e_{max} \leftarrow E_{max} $
8: $E_{max} \leftarrow \text{the } e_{max} \text{ largest elements from } \alpha$
9: push the remaining $c^{2/3}$ elements from α to level 1
10: $I \leftarrow \{\}$
The only difference is that lines 5-8 make sure that the size of both extraction
buffers is unchanged by insert.

I/O Complexity of Insert, Extract_min and Extract_max

As the I/O complexity of one push and pull is unchanged, the analysis of the I/O complexity of insert and extract in the Distribution Heap described in Section 3.1.3 on page 35, can be reused to analyze the I/O complexity of insert, extract_min and extract_max in the DIPD.

Algorithm 1	5 DIPD:	extract_min
-------------	---------	-------------

ent x

1: **if** extract buffer $|E_{min}| = 0$

2: $E_{min} \leftarrow \text{pull_min from level 1}$

3: $x \leftarrow$ smallest element from either I or E_{min}

4: remove x

To extract the smallest element, extract_min is changed to work on E_{min} .

The amortized I/O complexity of one insert, extract_min or extract_max is thus:

$$O\left(\frac{1}{B}\log_{M/B}\left(\frac{N}{B}\right)\right).$$

As a priority deque supports the same operations as a normal priority queue, plus the extra insert and delete operations, the lower bound on I/O complexity from the priority queues operations, can trivially be transferred to priority deques. The DIPD is therefore an *optimal* cache-oblivious priority deque.

Work Complexity of Insert, Extract_min and Extract_max

As shown in Section 3.1.3 the work complexity of a Distribution Heap can be computed in much the same way as the I/O-complexity. The changes made to create the DIPD do not change any of the dominating terms in the work complexity we computed for the Distribution Heap, and the amortized work complexity for inserting or extracting one element to/from the DIPD is thus $O(\log N)$. This implies that the DIPD is work optimal in the RAM model.

4.2.4 Construction

The DIPD can be constructed similarly to the technique used in the Distribution Heap, as described in Section 3.3.1 on page 56. First, all elements are sorted into a stream α . The levels in the DIPD are now filled one by one. First, the smallest elements from α is inserted in the min–buffers on the level, so that each of them is half full. Then, the largest elements from α are inserted in the max-buffers on the level. When all buffers on a level are filled, we proceed by filling the next larger level. This construction method is illustrated in Algorithm 16.

For each level, we need to access both small and large elements in α . This can be done by two scans, one scanning from the beginning of α , and one reverse scanning, beginning from the last element. The initial sorting of the elements will dominate the scanning, so the I/O complexity of constructing the DIPD containing N elements is sort(N).

Algorithm 16 DIPD: construction

Input: array A 1: sort(A)2: $L \leftarrow \text{first level}$ 3: $q \leftarrow |L|$. 4: while |A| > 0for all min–buffers B_{min} in L5: insert $q^{1/3}$ smallest elements in B_{min} from A 6: pivot of $B \leftarrow$ largest element in B_{min} 7: for all max–buffers B_{max} in L8: insert $q^{1/3}$ largest elements in B_{max} from A 9: pivot of $B \leftarrow$ smallest element in B_{max} 10: $L \leftarrow$ next level above 11: 12: $q \leftarrow |L|.$

In the DIPD construction the change is that in line 8–10 elements are also inserted into the max–buffers.

4.3 Summary

In this chapter, we have described, the DIPD and showed that it is possible to insert and extract elements from it, using amortized per element:

$$O\left(\frac{1}{B}\log_{M/B}\left(\frac{N}{B}\right)\right)$$

I/Os, $O(\log N)$ work, and that it is possible to construct a DIPD containing N elements in

$$O\left(\frac{N}{B}\log_{M/B}\left(\frac{N}{B}\right)\right),$$

using $O(N \log N)$ work. Like the Distribution Heap, the DIPD uses O(N) space. All in all these bounds on the priority deque matches the best bounds achievable for cache-oblivious priority queues.

Chapter 5

Measuring Performance

To err is human, but to really foul things up you need a computer.

— Paul Ehrlich

It is not an easy task to establish whether cache-oblivious algorithms are a viable alternative to the cache-aware techniques, using benchmarking. There are many factors involved, which makes it difficult to make well-founded conclusions. This chapter will outline our general approach for conducting benchmarks. In Chapter 6 we describe our specific approach for benchmarking priority queues. These tools and methods will be used for benchmarking in Chapter 7.

5.1 Previous Work

In order to find inspiration on how to conduct proper benchmarking of priority queues, we searched the literature for previous work on the subject. Here, we briefly summarize relevant methods described.

In 1986, Douglas W. Jones compares RAM-based priority queues in an empirical fashion [28]. In order to make measurements reliably, he makes sure that the computer he runs the benchmarks on is only used by him¹. Each program is run three times using a different random seed for generating data, but the same random seeds are used for each algorithm, in order to benchmark different algorithms with the same data sets, and thereby not favoring one algorithm by a "luckily" chosen data set. Furthermore, to determine the overhead imposed by the benchmarking itself, the benchmark is repeated with empty functions. This technique,

¹Something that was difficult at a time with scarce computer resources.

called *double-loop*, determines the overhead involved. Jones benchmarks his priority queue, by first constructing the data structure with a certain size, and then repeatedly calling insert followed by extract a number of times. By varying the size of the initial data structure, he obtains an average measure of insert and extract operations. This is an interesting approach, but for our cache-oblivious data structures this would not reveal much, as all the inserts and extracts would occur in the smallest level, utilizing the cache optimally. Even though this approach is not viable for our purpose, Jones' methodology for getting exact measurements, as described above, is still useful.

In 1992, Andrew M. Liao studied three priority queue applications, and used them for benchmarking priority queue data structures [36]. The benchmarks are run on relatively simple machines like, an IBM XT, with no cache or virtual memory, which is suitable for algorithms designed in the RAM model. Liao picks a similar approach as Jones, and runs each program three times with different random seeds. The final measure is obtained by taking the average of the three runs.

The behavior of cache-aware priority queues was studied by Brengel et al., in 1999 [13]. Here, four RAM-based and four cache-aware algorithms were compared. The strategy was to execute three different kinds of benchmarks:

- The first starts by performing a number of inserts, and afterwards does the same amount of extracts—this is done to measure raw I/O performance.
- Second, a mix of insertions and deletions were carried out for benchmarking I/O, and the speed of the CPU in the internal-memory part of the program. First, the priority queue is constructed with a number of elements. Following that, inserts occur with 1/3 and extract with 2/3 probability.
- Finally, an application benchmark using Dijkstra's shortest-path algorithm was carried out, in order to have a benchmark with real-life application data instead of random generated input.

Inspired by these previous approaches we now discuss how we are going to conduct our benchmarks.

5.2 Choosing Benchmark Tools

In order to make an objective benchmark of implementations, it is important to choose an appropriate approach and method of quantification.

First we want to find one single overall metric, that describes the overall performance of the implementations, by having some measurement of the total time an implementation needs to perform a task. Second it is also interesting to obtain measurements of the behavior of the different components of the memory hierarchy, when benchmarking cache-oblivious algorithms. It would be especially useful to get information on the amount of cache, TLB and page misses. To achieve this we have found two tools which can be used for this purpose: A library to read hardware counters, and a way of obtaining the number of page faults from the operating system. Finally, we look into profiling our implementations.

5.2.1 Timing

A common metric is *wall-clock time*, which is easy to handle. Wall-clock time is, however, sensitive to external events like interrupts and other processes running on the same computer. When using wall-clock time, it is therefore important to eliminate as many external factors as possible, by running on a machine with no other users and minimal amount of operating system processes.

Another common approach, is *CPU time*, which is the amount of active CPU time a program uses during a run. As this measurement only reports the amount of active time, for the process, this eliminates some of the uncertainties that occur in Wall-clock time. It is, however, important to notice that this method does not take external events, like paging, into account, because processes are not scheduled during a page fault. CPU time is, therefore, not suitable for comparing implementations that work on data so large that page-faults will occur.

We will, therefore, use wall-clock time as the metric of the overall performance.

5.2.2 PAPI

One approach for benchmarking is to have the hardware itself report information about e.g. cache misses. Most contemporary CPUs have *hardware counters* which can be instructed to register different events. However, the interface and the types of events which can be counted differ between different CPUs.

In order to program and read these hardware counters, a library called *PAPI* (Performance Application Programming Interface) [43] has been developed. This library aims at being a widely used cross-platform interface for accessing hardware counters on different architectures and operating systems.

PAPI has a high level C++ interface, which allows the user to start, stop and read counters from a specified list of events. Presently PAPI supports over 100 different events of which not all are supported on all architectures. Each event is defined as a constant with a suitable name—as an example all level 1 cache events are shown in the table in Figure 5.1. There are only a limited number of hardware counters available, so it may be necessary to run a program multiple times to measure all the events we want. Some events are overlapping, though: The number of Level 2 accesses are for example identical to the number of level 1 misses.

The PAPI library is well suited for revealing more detailed information about a program, than just the measured running time. With this library it is possible to measure the actual difference in the amount of cache misses generated by different programs.

PAPI-event	Description
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L1_TCM	Level 1 cache misses
PAPI_L1_LDM	Level 1 load misses
PAPI_L1_STM	Level 1 store misses
PAPI_L1_DCH	Level 1 data cache hits
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L1_ICH	Level 1 instruction cache hits
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L1_ICR	Level 1 instruction cache reads
PAPI_L1_TCA	Level 1 total cache accesses

Figure 5.1: Table of PAPI events for level 1 cache.

5.2.3 Page Faults

Because PAPI reads hardware counters directly from the CPU it cannot monitor *page faults*, as this is handled by the operating system. Of the platforms we have access to, the Linux and Solaris kernels both support reading the amount of page faults for each process. This is done through the /proc file system, which is basically a virtual file system consisting of direct access to internal kernel data structures. The manual page for proc (on Linux proc(5) and Solaris proc(4)) describes the functionality of these file systems.

Each running process has a unique id—a *process id* or *pid*. For every process, a directory named by the pid is created in /proc. This structure contains information on the process. Especially interesting for our purpose is the file stat on Linux, and the file usage on Solaris, which contains information about the amount of page faults the process has generated.

On both systems, there are two types of page faults: Minor and major. A minor page fault occurs when a new page is created. Minor page faults do not result in disk accesses. A major page fault occurs when a requested page is not in main memory, and has to be retrieved from disk. The number of major page faults corresponds to the number of I/Os incurred by page faults.

In implementations that by themselves access files stored on the file system, the accesses to theses files are not counted by this measurement. This implies that this measurement is only valid for implementations that do not themselves read/write data to/from the file system. So the measurement of page faults can not be used for external-memory algorithms, as they, themselves, handle the accesses to disk, and they will, therefore, not have page faults.

The information available from the /proc file system, thus makes it possible to obtain detailed information on the number of page faults.

5.2.4 Profiling with gprof

For the benchmarking, we are also interested in determining which parts of the implementations are responsible for the majority of the time spent. For this purpose we profile the implementations with the standard GNU profiler gprof, which is distributed together with the compiler we use (g++). The output of the profiling is enclosed in Appendix B.

Following the development of our implementations, we use the profiler to identify possible optimizations, but we also intend to use the profiler to identify the expensive parts of the implementations during the benchmarking. For these purposes we profile each of the operations in each of the implementations. We use a fairly large data size of 8,388,608 to get reliable measurements.

To ensure that the profiles are not influenced too much by the data sizes, we also made some experiments with a small data size, in order to verify the results.

5.3 Methodology

This section describes how we perform our benchmarks. Here we present which hardware and software we are going to use, as well as detailed information about how we set up the benchmarks.

We have developed a framework (included in Appendix A) that is used for benchmarking all implementations. This framework offers a simple and reusable way of parsing parameters and performing different types of measurements on the implementations. This means that whenever we want to implement a new algorithm, we do not have to worry about all of these issues, and we can concentrate on the algorithm we are implementing. More importantly, though, this framework ensures that all measurements are made in a uniform way, as the code performing the measurements is the same for all implementations.

5.3.1 Choosing Benchmark Data

To get reliable results, the dataset must be chosen carefully in order not to bias the results. On the other hand, the benchmarks should be reproducible and comparable between algorithms. For these reasons, we choose to generate data randomly, using the standard rand() function from the standard library stdlib.h. This is done using the same random seeds for every algorithm, in order to make the results comparable. The result is, that we will use a fixed data set, to get reproducible results.

5.3.2 Measurements

We benchmark with three different measures: Wall-clock time elapsed in microseconds, number of page misses generated, and the counting of a number of PAPI events. All the measurements are started and stopped as close to the actual operations as possible, and setup time such as loading of the program and reporting the results is therefore not included in the measurements. One thing that *is* included in these measurements, is the random generation of elements to be inserted during the benchmark, but this overhead is equal for all implementations, as we use the same benchmark framework.

Furthermore, we have created a special benchmark implementation called EMPTY, which executes the whole benchmark procedure, including generating the random numbers and the function calls to the implementation, but EMPTY does not execute anything and returns immediately. The measurements on EMPTY, therefore, show the overhead incurred by our benchmark system. The usage of EMPTY corresponds to the double-loop technique, as discussed in Section 5.1 on page 74. A common approach is to have each program do this double-loop, but we have chosen to have just the one program EMPTY, because all our programs use the same framework. When compiling a program with empty function calls, one should be aware of the risk of an optimizing compiler, removing unused functions. This should, however, not be a problem, because the random function has side-effects (it modifies the random sequence). We have verified that the random number generation is not removed during optimization, by looking at the assembler code produced by the compiler.

The measurements are obtained using the following techniques:

- **Time** This is measured with the C function call gettimeofday from the sys/time.h header file, which can report wall-clock time in microseconds.
- **Page faults** The values are read as described in Section 5.2.3. Only the amount of major page faults is measured, as this shows the number of I/Os performed.
- **PAPI** The PAPI library supports a large number of different events. We pick the most relevant for our use, from the ones describing caches and the TLB. Unfortunately, not all events are supported on all architectures we would like to use. This is an important factor when we decide which architectures to conduct benchmarks on.

5.3.3 Choosing Hardware Platforms

Our goal is to examine the practical value of cache-oblivious algorithms, by comparing them to cache-aware and RAM algorithms. For this task, it is relevant to conduct benchmarks on different kinds of computers in order to obtain results which are unbiased in regard to possible advantages/disadvantages that a certain model of computer could exhibit.

There are many different architectures and operating systems available today more than we have time to conduct benchmarks on. We must therefore narrow our choice of platform down to a few. Here certain practical considerations come into play: Which platforms are available and which support the tools we need? The important requirements are support for the PAPI library and the possibility of getting information about page faults from the operating system.

Based on which machines we have access to, only two meet these requirements. We have therefore chosen to conduct the benchmarks on the following two platforms:

	AMD Athlon 1000MHz	SUN UltraSparc-IIi 333MHz
System:	RedHat Linux 7.3	Solaris 2.8
RAM:	320 Mb	512 Mb
L1 Cache:	64KB	16KB
L2 Cache:	256KB	2048KB
Swap space:	5 GB	15 GB

The AMD Athlon CPU was picked over the more widely used Intel Pentium CPU because it supports reading of data TLB misses through PAPI, which the Pentium does not. To have an alternative platform for comparative benchmarks the Sun UltraSparc-IIi CPU was chosen because this platform also supports PAPI and reading of page faults.

It is important to note that both CPUs have a fairly slow clock speed, and that the fastest CPUs in the same families have up to two or three times faster frequencies. These two computers are, as mentioned, selected due to their capabilities for performing our benchmarks and because we did not have access to faster CPUs that could have made the same benchmarks. As the development of relatively faster CPUs continues, the importance of the memory use will also be higher in even faster CPUs, so the results we achieve on these fairly slow CPUs, will give a conservative description of the importance of the memory use. In the remaining text we refer to the AMD Athlon as "AMD" and to the Sun UltraSparc-IIi as "Sparc". The table in Figure 5.2 summarizes the PAPI events we use on the architectures.

Platform	PAPI event	Description
	PAPI_L1_DCM	Level 1 Data Cache Miss
	PAPI_L1_DCA	Level 1 Data Cache Access
AMD	PAPI_L2_DCM	Level 2 Data Cache Miss
	PAPI_L2_DCA	Level 2 Data Cache Access
	PAPI_TLB_DM	Data-TLB Miss
	PAPI_L1_LDM	Level 1 load misses
	PAPI_L1_STM	Level 1 store misses
Sparc	PAPI_L1_ICH	Level 1 instruction cache hits
	PAPI_L1_ICA	Level 1 instruction cache accesses
	PAPI_L2_TCM	Level 2 cache misses

Figure 5.2: Table of PAPI-events we used in the benchmarks.

5.3.4 Configuring the Benchmark Computers

As we are interested in making benchmarks of the behavior on all levels of the memory hierarchy, we need to execute some benchmarks where the input is too large to fit in main memory. These benchmarks will, for cache oblivious and RAM algorithms, incur a number of page faults. This is also where the external-memory algorithms will be relevant. As most contemporary computers have a large main memory, these benchmarks will require very large datasets, and therefore the benchmarks will have a very long running time; too long to be applicable in our benchmarking.

Another problem with these large datasets, is that we might run out of addressspace. As described in Section 3.5, a 32-bit computer can only address 4GB of memory. The way the Distribution Heap is constructed, however, requires space at least equal to four times the number of elements. On a 32-bit CPU like the AMD Athlon running Linux, the amount of memory available for a user-space program is approximately 2GB. With this limit, the largest possible size of a Distribution Heap would be less 512KB. This is however not an unusual amount of main memory to have in a contemporary computer. This implies that the amount of I/O needed to the disk will be fairly limited, and this will make it impossible for us to benchmark the implementation's behavior when data is larger than the main memory.

We handle these two problems by reducing the amount of available main memory. This can be done in two ways: First, it is possible to boot computers with less main memory than physically available. Second, on many operating systems, it is possible to lock allocated memory, so that it stays in the main memory. This can be used to make a simple program that allocates and locks parts of the main memory, so that other applications cannot use it. The first option requires rebooting the computer over and over. When using the second option, it is easy to start and stop. We have therefore chosen the second.

For this we use a small program called blockmem², implemented by Jacob Poulsen, for locking data in main memory. On the Linux machine we leave 64Mb free, but on the Solaris we are only able to restrict the free memory to 112Mb—this is due to the way Solaris handles allocated memory. This should, however, not pose a problem as our largest datasets are larger than 112Mb, and therefore force the system to swap memory pages. It will however make the comparison of the two platforms more complicated. This is, however, not a problem, because we are not interested in the absolute difference in performance between the two platforms, but rather to see whether they exhibit the same behavior when running the algorithms.

5.3.5 Validity

Most modern computers run multitasking operating systems. When benchmarking a single program, this introduces uncertainty because we do not have control over

 $^{^2} This$ software is released under GPL [24], and can be downloaded from www.dunkel.dk/ thesis.

everything that happens in a computer: The measurement of wall-clock-time can be disturbed by scheduling of processes, occurrence of interrupts and network traffic.

Furthermore, the scheduling of other processes can result in the level 1 and 2 caches getting flushed or filled with data from the other processes. This will cause cache misses, once the process is scheduled again, and result in inconsistent results. It is therefore important to eliminate as many external factors as possible, while still making the benchmarking procedure practically possible.

A lot has been written on the subject on how to obtain valid benchmark results. One example is an article by Koening and Moo [31], in which they describe how to obtain very precise measurements in a reliable fashion. Their approach is to determine on an acceptable level of variance, e.g., requiring that 90% of the measurements must be with 10% of the median of all measurements. This could be achieved by making 20 measurements, removing the smallest and largest element and checking that the remaining elements are within 10% of the median. If this is not the case, more measurements must be obtained until the limit is reached. If it is impossible to obtain the wanted variance, this procedure never ends—in this case the requirements must be relaxed.

This kind of careful measuring is appropriate when very exact measurements are needed—on the scale of microseconds—where external influences have a great impact. For our purpose, the vast majority of our benchmarks will run for several seconds, and most of them even minutes, so we can relax this very strict approach. In practice, an often used solution is to conduct a couple of runs of a program and use the average as the measure. We therefore repeat each run 5 times and take the average. This is chosen to make sure that deviations in one result, will have a fairly small influence on the overall result.

In addition, each of the 5 runs is performed with different random seeds, which minimize influence of a biased dataset. A more precise result could be obtained by running each benchmark as the only process on a computer, and to repeat it many more times. On the other hand, as we are mainly interested in asymptotic differences between programs, small differences does not matter much.

When comparing implementations, it is important to note that some of the differences might be the result of different code qualities. It is reasonable to assume that the implementations we have found in well-known libraries as, e.g., Silicon Graphics STL [48], are very efficiently implemented, whereas we expect that it will be possible to find improvements in our implementations. It is therefore important to remember that the differences found when benchmarking can be due to different coding quality, as well as actual differences between the algorithms.

All in all, the benchmarks we conduct, using the methods described here, should be sufficient to reveal valid facts about the algorithms and display differences in the different implementations.

Chapter 6

Preparing Benchmarks of Priority Queues

Errors using inadequate data are much less than those using no data at all.

- Charles Babbage (1792-1871)

In Chapter 5, we described the general procedure we are going to use for benchmarking. In this section, we describe the priority-queue specific parts of the benchmark strategy. This includes the preparations needed to make fair priority queue benchmarks.

First, we describe the benchmarking strategy for priority queues. Second, we find other implementations of priority queues to compete with the cache-oblivious implementations. Finally, we describe practical issues regarding the benchmarks.

6.1 Sequences of Data

To get a fair idea of how the different implementations perform, we need a series of benchmarks that try to cover all aspects of the performance of the implementations. Therefore, we need different sequences of data to cover all cases. We are interested in comparing each function in the implementations, so we first choose a measurement which benchmark one function at a time. Secondly, we would like to have benchmarks that summarizes the overall performance of the entire implementation.

In the following text we use abbreviations of each of the the operations: I = insert, E = extract and C(N) = Construction of a priority queue with N elements.

The simplest sequence that summarizes a general use of a priority queue, is a sequence of inserts followed by the same number of extracts. This use is, e.g., found when a priority queue is used for sorting [19].

Other sequences might, however, result in significantly different results. An example of this, was experienced by Sanders [46]. He initially benchmarked his priority queue with the sequence:

 $(IEI)^n$,

where n is the number of repetitions. The theoretical analysis did, however, show that some of the algorithms might perform worse when the number of inserts becomes very large compared to the maximum size of the priority queue. He, therefore, also tried the more general sequence called *long operation sequence*:

 $(I(EI)^s)^n \ (E(IE)^s)^n,$

where n and s are the number of repetitions.

Contrary to the theoretical result indicating worse performance, some of the priority queues actually performed better when s was increased [46]. This is caused by the fact that when random numbers are inserted, and the smallest elements are removed from the priority queue, the numbers left in the priority queue become larger and larger over time. Due to the structure of the priority queues studied, the largest elements will be stored in the larger and most expensive levels of the priority queue. After a number of element insertions and extractions, most elements inserted in the priority queue will be smaller than the majority of the elements contained in the priority queue. The result of this is that most elements are inserted, and shortly after extracted again. Priority queues that store these elements in small and fast structures, will then experience an improvement in performance when s is increased. As both cache-oblivious algorithms store data in buffers of increasing size we expect that they has the same property of improved performance if s is increased, so we also run benchmarks following the sequence of operations described by Sanders.

This locality of data is however not necessarily present in real uses of priority queues. One well known use of priority queues, is Prim's algorithm for finding a Minimum Spanning-Tree in a graph, and Dijkstra's algorithms for Single-Source-Shortest-Path¹ [19]. These algorithms, however, both have the property that the priority of the elements that are inserted are guaranteed to be larger than the last element that was extracted. The smallest element in the priority queue is monoton-ically increasing over time, which is the opposite behavior compared to the long operation sequence.

We would like to use a sequence of operations that simulated this behavior, where elements are larger than the previous extracted elements. We therefore make

¹This algorithm require a decrease-key operation in the priority queue, which is not directly available in all priority queues

a sequence where we start by inserting n random elements. Afterwards we extract a minimal element and insert a new element that is larger than the minimal element. This sequence of data has the effect that the smallest elements in the priority queue are monotonically nondecreasing, so that once an element has been extracted with value x, all elements added to the priority queue are larger than x.

The table in figure 6.1 summarizes which sequences of data we use for the benchmarking.

Sequence	What will this sequence reveal performance of
I^n	insert
E^n	extract
C(n)	priority queue construction
$C(n) E^n$	simulation of priority-queue sort
$I^n(IE)^nE^n$	monotomical nondecreasing minimal elements
$(I(EI)^s)^n \ (E(IE)^s)^n$	Sanders long operation sequence

Figure 6.1: Summary of the sequences used.

It is important to note that a direct comparison of algorithms based on only one of the operations $(I^n, E^n, \text{ or } C(n))$ is not necessarily fair. Some algorithms do more of their work during insert than extract and vice versa. On the other hand, it is still interesting to look into the performance of the single operations, as it will be easier to interpret the results.

6.2 Data Sizes

We benchmark each of the abovementioned sequences with a number of different sizes. We have selected the smallest size to be small enough to fit into the first cache level on our benchmark computers. The largest size is chosen so large that all implementations access disk. We perform measurements in the range [1024–33,554,432]. The size will be doubled each time, so the sizes benchmarked will be: 1024, 2048, 4096, 8192 ... 33,554,432.

The measurements do, in some cases, take very long time—especially when the RAM implementations are run on datasets so large that swapping occurs. Therefore, we have made our benchmark system in such a way that it monitors the execution time, and if a single measurement takes more than approximately one hour, it is terminated and the following larger measurements are not performed. This is necessary to make sure that the benchmarks are available within a reasonable time.

6.3 Types of Elements

Our benchmarks are mainly performed using simple integers. Because the implementations copy and compare the elements a number of times during a run, the performance of the implementations is expected to differ when the elements are larger than simple integers. This is tested by running the benchmarks with a complex data type, constructed for the purpose. Inspired by a commonly used benchmark from the database world called *the sorting benchmark* [41], we choose a complex type with a size of 100 bytes and a key of 12 bytes.

In C and C++, most allocations of memory space will have to use additional space for bookkeeping, as, e.g., described by Bentley [10]. In practice, this means that even simple data types, like integers which the programmer could assume was of size 4 bytes, actually uses 16 bytes on contemporary computers. Bentley has made a small program spacemod.cpp which can estimate the actual space used when creating a new object with new. Using this, we found that one instance of our large data type of 100 bytes uses 104 bytes, but that an entire array only uses 8 extra bytes. At any given point our programs allocate fewer than 10 arrays, so this overhead will not have any influence on the behavior of the benchmarking of the large data type.

6.4 Choice of Compiler

To get good and fair benchmark results, it is important to choose the compiler in such a way that the benchmarks can be compared. The first requirement is, therefore, that the compiler must be available on both our benchmark platforms. Furthermore, the most relevant results are achieved when the compiler produces fast optimized code, as some algorithms and implementations might be easier to optimize than others. As any user that relay on good performance naturally will optimize his programs, this would make a comparison of unoptimized implementations of less practical relevance. Finally, the compiler should be as widely available as possible, so that other people are capable of reproducing the benchmark results.

Based on this, we have chosen to use the GNU C++ compiler g++ [23]. To get the best optimization we have retrieved the newest version, which is 3.2, and optimize all implementations with the highest level of optimization -03.

However, we have experienced problems with the compilations on the Sparc. By trial and error, we discovered that the compiler, for some reason, produced erroneous executables when functions was inlined. We therefore had to disable this on the Sparc, using the option -fno-inline-function. By a couple of simple runs we have determined that inlining functions improves the performances by approximately 5–10%. This is not a big problem in practice, because we are mainly interested in the relative difference between the implementations on each of the benchmark computers—not directly comparing them.

6.5 Choosing Competitors

When benchmarking implementations, it is important to find relevant competitors that can be used as a reference when determining whether the new implementations offer an improved performance. In this case it is important that the competitors are both standard RAM implementations, as well as cache-aware implementations. This section describes the competitors we have chosen and how we have ensured, that the comparisons are as fair as possible.

6.5.1 RAM-model Priority Queues

STL Heap

The obvious RAM-model competitor to use is STL, which is a very widely used library with many different algorithms, including a priority queue. Because STL is a part of the C++ standard, it is included as default with many compilers. This makes it available and easy to use for all C++ developers. We therefore use the STL heap for comparison. For our benchmarks, we will use the SGI implementation [48, 49] of STL.

Boost Heap

Apart from the STL, there is also an alternative C++ library, called Boost, which also has a number of priority queues. They are, however, currently only in beta release [12]. Their main focus has been to add additional facilities to the data structures, such as accessing elements in the priority queue through iterators and/or to change the priority of elements in the priority queue.

This implies that most of the implementations have a non-optimal asymptotic complexity in the RAM-model. The author of the different algorithms has pointed out that the fastest of the implementation is boost::priority_queue, and this is, therefore, the only one include in our benchmarks.

To use the Boost priority queue in our benchmark system, we had to add a heap constructor. This is simply done by repeatedly inserting the elements one at a time. This implies that the heap construction has the non-optimal work bound $O(N \log N)$. Consequently, this operation is expected to perform worse than the STL heap, but LaMacra and Ladner [33] showed, that the cache usage of this method is better. By comparing the two RAM-model implementations, we are able to determine whether this is also the case on our benchmark computers.

6.5.2 Cache-Aware Priority Queues

LEDA-SM

LEDA is a library with efficient data types and algorithms for many algorithmic problems, including graph- and network-problems, geometric computation and combinatorial optimization. A spin-off project called *LEDA-SM* (*LEDA for Secondary Memory*) for is working towards incorporating external-memory algorithms into LEDA. It currently contains a implementation of an external-memory priority queue, called *the Radix Heap* [35]. This priority queue is the result of experiments

with several different priority queues, performed by Brengel et al. [13]. Their result was that the radix heap was the fastest, and therefore this algorithm is used in LEDA-SM.

To compare our implementation with the radix heap, we have made an STLlike interface to the radix heap, so we are able to use our standard benchmarking system. There are, however, some differences that limit the possibilities of this interface: The biggest problem is that the radix heap is designed to work only on key/value pairs, and the key has to be an integer. This limits the interface to only work with elements of integer type, which implies that we are not able to perform benchmarks with the large data type on the LEDA-SM implementation.

The LEDA-SM implementation also requires that elements must have both key and value, so our STL-like interface inserts a dummy value which is never used, and stores the integer element in the key. This solution has the disadvantage that the radix heap will use extra space, and run out of memory faster than if a proper STL compliant version had been implemented.

The radix heap has another major problem: It uses knowledge of the minimal element to determine the behavior of the data structure, and it requires that the minimal element in the priority queue is nondecreasing. This means that whenever an element with a key is extracted, it is not possible to insert elements with a smaller key. This is, however, exactly the behavior of the *long operation sequence* described earlier. Therefore we are not able to run this benchmark sequence on the LEDA-SM priority queue.

Finally, we do not have access to a version of LEDA for Solaris, and are therefore not able to run any benchmarks of this external-memory priority queue on the Sparc benchmark computer. Despite these limitations, we still include the LEDA-SM priority queue in our benchmarks

Sequence Heap

Sanders [46] described and implemented another priority queue. This is neither an external-memory nor a cache-oblivious algorithm, but is an adaptation of an external-memory algorithm, which also use the cache in an optimized way—in other words a *cache-aware* algorithm. This priority queue is called the *sequence heap*, and we include this priority queue in the comparison with our cache-oblivious priority queues.

Since the sequence heap is a cache-aware implementation, it is possible to tune it to perform well on a specific computer. Sanders, however, observed that one specific parameter setting performed well on all the computers he had access to. Among these, are computers that are fairly similar with our benchmark computers, so we have chosen to use these parameters as the parameters for our benchmarks, too.

One could argue, that when using this value, the algorithm is cache-oblivious, because it does not know the exact sizes of the memory levels, but the term cacheaware is still the most suitable. The difference is, that Sanders implemented this algorithm to be parametrized with the sizes of the levels. During his work, he then discovered that most comtemporary computers actually performs well with the same set of parameters for his algoritm.

The version we use in our comparisons, is an adaptation of the implementation used in [46]. This implementation was not made to conform to the STL standard, so we had to make small changes to this, too. The STL standard states that priority queues operate on elements, whereas the sequence heap implementation works with key-value pairs. We simply removed the value from all functions in the sequence heap, so that all information is stored in the key. This was done to achieve a fair comparison of the different implementations. Furthermore, the implementation assumes that the Key-type has the functions $\langle, \rangle, \geq, \leq, ==$, and !=. defined, whereas the STL standard requires that the user can change the comparison by changing a template parameter. We have adapted the implementation to work as expected in our benchmark environment, but the altered implementation does not conform to the STL standard.

6.6 Practical Issues

We discovered—as mentioned in Section 3.1—a missing detail in the description of the Distribution Heap [6] very late in the process of writing this thesis, by corresponding with two of the authors Arge and Holland-Minkley. This misunderstanding was first cleared two weeks before our deadline, which meant that we had to adapt our implementation to this change late in the process. As the benchmarking of the Distribution Heap implementation using the system described in Chapter 5 is quite time consuming, we have had to relax the requirements to the benchmarking of the Distribution Heap implementation. We would-as described in Section 5.3.5—have liked to repeat all measurements five times to ensure that the results where not influenced by external events. For the benchmarks of the Distribution Heap, however, we have not repeated the measurements. By looking at the results of the benchmarks of the other implementations we have determined that the variance in these are less that 5% of the results in all cases, which is too small to influence any of our conclusions. We also described benchmarks with a 100 byte large data type. These benchmarks naturally take more time, so we have only been able to perform some of these benchmarks.

We combine the benchmark results for each implementation in graphs which show measurements of one specific sequence of operations. The results from all implementation are shown in the same graph. The x-axis show the number of elements while the y-axis show the measured results. Furthermore, we have added vertical dotted lines to show the sizes of the level 1 and 2 caches as well as main memory.

In the graphs we use short names for the different implementations. The table in Figure 6.2 shows the interpretation of these short names.

We benchmark 13 different implementations on two different computers taking 16 measurements for seven different sequences of operations with two data types. Needless to say, this generates huge amount of results, which is more than we can possibly present in this thesis.

In order to reduce the amount of figures presented we have made a thorough analysis of all data produced by the benchmarks and identified the interesting observations and results. Based on this analysis we have selected graphs which illustrates each of these points. All graphs generated by our benchmarks can be found in Appendix B.

Short Name	Description
DISTHEAP	Distribution Heap. Implementation of the opti-
	mal cache-oblivious algorithm. Uses Funnel Sort
	to sort and lazy_select to find the median.
DISTHEAPSTL	Distribution Heap. Subroutines avaiable from the
	SGI STL are used for sorting and median finding.
DISTHEAPALTRB	Distribution Heap with the alternative rebuilding
	using scan instead of extract. Here Funnel Sort is
	used for sorting and lazy_select for median
	finding.
DISTHEAPALTRBSTL	Distribution Heap with the alternative rebuilding
	using scan instead of extract. Here SGI STL is
	used for sorting and median finding.
FUNNELHEAP	Funnel Heap. Implementation of the optimal
	cache oblivious algorithm. Uses Funnel Sort for
	sorting.
FUNNELHEAPSTL	Funnel Heap. Uses SGI STL for sorting. Since
	sorting is only used in Funnel Heap construction
	this implementation is only relevant in sequences
	that performs construction.
FUNNELHEAPSTL	Funnel Heap. As we will describe in section 7.3
	on page 103 we get the idea for an possible op-
	timization of the Funnel Heap. This implemen-
	tation uses this alternative sweeping method and
	uses Funnel Sort for sorting.
FUNNELHEAPATLSTL	Funnel Heap. Uses the alternative sweeping
	method, and uses subroutines from the SGI STL
	for sorting.
STL	SGI STL RAM-model heap implementation.
BOOST	Boost implementation—called p_queue [12]—of
	the RAM–model heap.
LEDASM	LEDA-SM implementation of the external mem-
	ory algorithm radix heap.
SEQUENCE	Sander's implementation of sequence heap [46].
EMPTY	A dummy priority-queue, where all operations
	are empty. This implementation is used to dis-
	play the overhead incurred by our benchmarking
	system.

Figure 6.2: Table showing the interpretation of short names used for our implementations.

Chapter 7

Performance Investigation

In theory, theory and practice are the same. In practice, they aren't.

— unknown

This chapter presents our empirical results on performance of the different priority-queue algorithms. The purpose of the benchmarking is to obtain results which investigate whether the theoretical advantages of cache-oblivious algorithms are also present in practice. To answer this question we study the following issues:

- The main advantage of cache-oblivious algorithms is that an optimal algorithm theoretically incurs an asymptotically optimal number of cachemisses, on all levels in the memory hierarchy. We are interested in examining how these assumptions about the ideal-cache model relate to real life.
- An important issue is whether the cost of the extra amount of work that might occur in cache-oblivious implementations, is smaller that the gain from asymptotic optimal use of the cache. As the cost of, e.g., a level one miss, is in the order of 10–20 cycles, the extra work spent to avoid cache misses might simply be too large. This is especially relevant for the data sizes that are small enough to fit in the smaller levels of the memory hierarchy, where the simpler RAM-based algorithms might have an advantage, due to their potentially less work.
- The last level in the memory hierarchy, is the disk. The special interest of this level is that external-memory algorithms are specifically designed to be optimal on this level. We are interested in determining whether the cache-oblivious implementations are able to compete with the external-memory implementations, when working on datasets so large that data need to be stored on disk.

We start this chapter by investigating which influence the choice of a sorting method has on the performance of the implementations. Subsequently, we look at the difference in the amount of work, and I/Os, each of the implementations perform, to determine whether the asymptotic fewer cache-misses incurred by the cache oblivious implementations are worth the potential extra work. Following that, we look at the measured TLB-misses to verify our theoretical result, which showed that the TLB will be used in an asymptotically optimal way by optimal cache-oblivious algorithms. Finally, we describe the overall performance of the implementations, and summarize our benchmarking of the priority queue implementations.

7.1 Sorting Method

One of the most significant results of the benchmarks, is a very large influence of the sorting method used in the algorithms. We therefore start by investigating this issue. The first result we want to examine is the wall-clock time measurements, because these give a good overall indication of the performance of the different algorithms. Figure 7.1 shows the measurements of wall-clock time during insertion on the AMD, and Figure 7.2, on the Sparc. It is clear that there is a large and consistent difference between the two sorting methods: Funnel Sort and STL. In DISTHEAP and DISTHEAPALTRB, the performance is radically slower than DISTHEAPSTL and DISTHEAPALTRBSTL. In most cases, Funnel Sort more than doubles the execution time of the implementations. This implies that Funnel Sort performs much more work than standard STL sort.

Another point worth noticing, is that all Distribution Heap implementations have an interesting development, that occasionally the measurement of the next and larger data size has better performance than the previous. This is caused by the way the levels are created during rebuilding, and the improvement occur each time the Distribution Heap adds another level. This causes the smallest level to be of the minimal size, which results in better performance. When the size is increased further this improvement disappears, as the size of the smallest level is also increased.

The very high cost of sorting is further illustrated by looking at the profiling of the implementations, which shows that DISTHEAP spends 95% of the time sorting, whereas DISTHEAPSTL only uses 63%.

During construction of Distribution Heaps, the cache-oblivious properties of Funnel Sort come into play, as seen in Figure 7.3. Here, we see that with large amounts of data the construction of the Distribution Heap is faster when using Funnel Sort, than when using STL Sort. This is because the Distribution Heap construction sorts all elements in one sort operation. In all other usages, the sorting is performed on smaller parts of the data sets. This indicate that our Funnel Sort implementation is much slower than the SGI STL implementation, when sorting small data sets. When the data size exceeds the amount of main memory, this behavior is reversed, so that our Funnel Sort implementation becomes faster.



Figure 7.1: Wall-clock time for inserts on AMD.



Figure 7.2: Wall-clock time for inserts on Sparc.

Also noticeable, in Figure 7.3, is that LEDA-SM has extremely poor performance on the smallest datasets. This is explained by the fact that LEDA-SM creates a file on the local file-system, when the priority queue is initialized. As this initialization is part of the priority queue construction, this is reflected as a large startup cost when constructing a LEDA-SM priority queue. LEDA-SM was, however, not designed to be used for small amounts of data, so this behavior is acceptable.

In Section 6.5.1 on page 87 we mentioned that BOOST did not offer a construction method as opposed to STL. Instead, we made our own by repeated inserts. This construction method does not offer optimal work complexity. However, LaMacra and Ladner [33] have previously showed that this method actually performs better due to its high locality of data. The results on Figure 7.3 supports this, and is an indication of the fact, that the heap algorithm's insert operation has a high locality of data, when used for construction. In fact, the STL construction is significantly worse than constructing an empty priority queue and then inserting the elements one by one.



Figure 7.3: Wall-clock time for construction on AMD

We have used Funnel Sort to sort elements in our implementations. This sorting algorithm is optimal in the ideal-cache model, but—as we have shown—has a high amount of work in practice. An interesting subject for further investigation is therefore to find possible improvements to cache-oblivious sorting.

7.2 Locality of Data

We now turn our attention toward the cache performance of the algorithms, in order to examine whether it is feasible to do extra work in order to get better cache usage. The cache-oblivious algorithms are designed to exhibit better locality, which should be reflected in the number of cache misses.

A prime example of the effect of *not* taking the cache into account, is seen in Figure 7.4, where the two RAM-based implementations, STL and BOOST, begin to incur a lot of cache misses, once the data sizes exceed the level 1 cache size.



Figure 7.4: Level 1 cache misses for extracts on AMD.

Looking at the number of accesses to the first level of cache during the extract operation in Figure 7.5, there are several interesting observations: The cacheoblivious algorithms are notably worse than all the competitors. There is a big difference in the number of level 1 accesses, depending on the sorting method used. This is, however, not reflected in the number of level 1 misses showed in Figure 7.6. Here, the difference between these two measures is much less apparent. This indicates that a lot of CPU work is done for the Funnel Sort algorithm, but that it has a high locality of data. This tendency is further enhanced in the level 2 cache, as seen in Figures 7.6 and 7.7. Also very apparent, is that all versions of the cacheoblivious Distribution Heap incurs more cache misses in both level 1 and level 2 cache, than the RAM-based algorithms. Furthermore, the Distribution Heap has a higher growth rate than the other algorithms. This indicates that this algorithm has a high workload, due to the rebuilding procedure.



Figure 7.5: Level 1 cache accesses for the complex sequence on AMD, zoomed.



Figure 7.6: Level 1 cache misses for the complex sequence on AMD (equal to level 2 accesses).



Figure 7.7: Level 2 cache misses for the complex sequence on AMD.

The low number of level 2 misses incurred by the implementations using Funnel Sort, is, however, not reflected in the wall-clock time measurements in Figure 7.8, which illustrates that the cache-oblivious implementations perform a lot of CPU work, and that this makes them slow. It is also noticeable that both RAMbased algorithms never finishes the last measurements within the time frame of the benchmark system. This indicates a huge increase in wall-clock time, once the data set grows close to the amount of available main memory. On the contrary, the cache-aware implementation SEQUENCE performs really well—it is almost unaffected by having to access the disk. Furthermore, this implementation is very competitive with the RAM-based algorithms, in the small levels of memory, which indicates a very little work in SEQUENCE. Similar results are achieved with the 100 byte large data type, as seen in Figure 7.9.

In fact, SEQUENCE performs better than STL, for all, but the insert operation. One example of this, is seen in Figure 7.10. This indicates that it can be viable to try to optimize algorithms for cache usage, but that the work involved should be small too. The external-memory algorithm LEDA-SM also proves its worth, as seen on the results of priority-queue sort, in Figure 7.11.

Looking again at Figure 7.1 on page 94 we observe that the Funnel Heap is slower than both the RAM-based and the cache-aware algorithms, at performing insert. This is explained by the fact that the large majority of work in this algorithm happens during insert. It is also reflected in the results for extract, which are very fast, as seen in Figure 7.12. The reason why the Funnel Heap is slow during insert, is that it does most of its work during the *sweep* operations.



Figure 7.8: Wall-clock time for the complex sequence on AMD.



Figure 7.9: Wall-clock time for the complex sequence on AMD with the large data type.



Figure 7.10: Wall-clock time for the monotonic non-decreasing sequence on AMD, zoomed.



Figure 7.11: Wall-clock time for priority-queue sort on AMD, zoomed.



Figure 7.12: Wall-clock time for extracts on AMD

If we turn our attention towards the amount of page faults incurred during extract, we discover some interesting properties. Looking at Figure 7.13 we see that the amount of page faults incurred by the RAM-based algorithms never finished within the time frame of the benchmark system. All the cache-oblivious versions of the algorithms also experience a huge increase in the amount of page faults. Especially, the Distribution Heap gets many more page faults than the Funnel Heap, which is caused by the need for rebuilding. Also, LEDA-SM perform the same number of page faults as EMPTY. This, of cause, is explained by the fact, that an external-memory algorithm itself handles paging, and therefore circumvent the virtual memory system. Again, the cache-aware SEQUENCE is the fastest, but the Funnel Heap is very close on the last measurement.

Interestingly, when looking at the number of page faults during insert, as shown in Figure 7.14, the picture is changed. Here, all versions of the Distribution Heap actually incur more page faults than the RAM-based algorithms. Again, as an effect of the expensive rebuild procedure. If we zoom this figure on the y–axis, as in Figure 7.15, we see that the Funnel Heap actually is competitive with the cache-aware algorithms. It is noticeable that STL, BOOST, and the Distribution Heap implementations, incur more than 25,000 page faults, where the Funnel Heap implementations only incur around 25. The competitors that actual try to minimize the number of page faults: LEDASM and SEQUENCE, has approximately 125 page faults on the largest dataset. This indicates that the advantage of minimizing the I/Os begin to take effect.



Figure 7.13: Page Faults for extracts on AMD



Figure 7.14: Page Faults for inserts on AMD


Figure 7.15: Page Faults for inserts on AMD, zoomed.

7.3 Rebuilding and Sweeping

The Distribution Heap needs to perform global rebuilding, and our intuition tells us that the cost of this might have a significant impact on the performance.

Returning to the wall-clock time of insert, in Figure 7.1 on page 94, we see that the implementations with the alternative rebuild procedure are faster than the implementations based on extracting elements using pull. To investigate this further we turn to the results of the profiling:

	DISTHEAP	DISTHEAPSTL	DISTHEAPALTRB	DISTHEAPALTRBSTL
Global Rebuilding	80.1%	82.2%	0.2%	0.9%
Emptying	79.9%	81.4%	_1	_1

The table above shows that global rebuilds account for approximately 80% of the time in the DISTHEAP and DISTHEAPSTL implementations, which is much more than we would have expected. To investigate this further, we have computed

¹The time spent here was too small to measure.

the amount of time spent extracting the elements from the old Distribution Heap, which is the first part of a global rebuild. This is shown as *emptying*, in the table. The astounding result, is that approximately 99% of the cost of rebuilding in DISTHEAP and DISTHEAPSTL, is spent extracting elements from the old Distribution Heap. On the other hand, DISTHEAPALTRB and DISTHEAPALTRBSTL, which remove the elements from the old Distribution Heap, and sort them, only uses 0.8% of its running time on global rebuilds.

This shows, that the version that sorts the elements through extraction from the heap—although optimal in theory—is very expensive in practical applications. The method of extracting elements, using a scan and sorting the whole array, is much faster in practice. Looking again at Figure 7.14 on page 102 there are, however, indications that once the data set is larger than the main memory, the alternative rebuild method eventually becomes slower. This is because page faults are very expensive and the extra work used can therefore be paid for by reducing the number of page faults.

This discovery gave us the idea to add an alternative to the Funnel Heap, which does not use the heap itself to extract the elements during the *sweep* operation. We therefore profile the Funnel Heap:

	FUNNELHEAP	FUNNELHEAPNEWSTL
sort	$0\%^{2}$	55.7%
emptying	51.4%	8.9%
reinserting	7.6%	13.6%

Here the cost of emptying links during a *sweep* (denoted *emptying* in the table), is rather high. This indicates that it might be possible to improve the performance by changing the way the elements are extracted, similar to the version of the Distribution Heap, which removes the elements in unsorted order, and subsequently sort them.

Based on these findings, we implemented an alternative version of the Funnel Heap. This new version scans the data structure, and copy all elements to a separate array and then sort this. This gives us two new algorithms, which are included on the graphs: FUNNELHEAPALT and FUNNELHEAPALTSTL.

7.4 TLB and the Virtual Memory Hierarchy

In Section 2.3.6 on page 23, we showed that a cache-oblivious algorithm—in theory—achieves an asymptotically optimal number of TLB misses. We are interested in verifying this result, by comparing the behavior of the cache-oblivious implementations with the other implementations. Especially, the cache-aware implementation SEQUENCE is interesting, as it is tuned to perform optimally in the levels of the ordinary memory hierarchy, but not the virtual memory system. We

²This version does not sort during insert.

investigate whether this is reflected in practice, by looking at the TLB, which is the first level in the virtual memory system.

Figures 7.16, and 7.17, show the number of TLB-misses per element for two different sequences of operations. They show that the two RAM-based algorithms (STL and BOOST) at some point begin to incur a lot of TLB misses, which is due to their poor locality of data.



Figure 7.16: TLB misses for monotone nondecreasing sequence on AMD.

In Figure 7.16, the Funnel Heap is the first algorithm to get TLB misses. This is due to the expensive *sweep* operations during insert, but this behavior eventually flattens out and becomes better than the RAM-based algorithms, although it is still a lot higher than the other cache-oblivious and cache-aware algorithms. This behavior is not present in Figure 7.17, as sweeps only occurs during insert.

The behavior of LEDASM, in Figure 7.17, is a consequence of the fact that this algorithm first creates a large file for its own memory page handling—this is relatively expensive for small data sets. It, however, also incurs a higher number of TLB misses on larger datasets, which show that it does not access the data stored in memory, in a fashion which exploit locally. This is explained by the fact, that the focus of LEDA-SM is on minimizing page faults.

In Figure 7.18, we have focused on the best performing implementations in regard to TLB misses during priority-queue sort. It shows that the Funnel Heap implementations and SEQUENCE have roughly the same minuscule, amount of TLB misses. As these two algorithms are very similar in design, this substantiate the interpretation that the good TLB behavior is a result of good locality in data access



Figure 7.17: TLB misses for priority-queue sort on AMD.

patterns. Both RAM-based algorithms have a much larger growth rate than the other algorithms, which is as expected for algorithms with little locality. The Distribution Heap versions with STL Sort, incur more TLB misses than the other cache optimized algorithms, which is explained by the use of STL Sort. Nevertheless, all of the cache-oblivious, and the cache-aware SEQUENCE implementations, exhibit good TLB usage. These are also better than LEDA-SM which is only tuned for one memory level. Therefore, it appears like good TLB usage can be achieved through good locality of data. This is the cache for both cache-aware and cache-oblivious algorithms.

These results verifies, to some extent, our theoretical reasoning that the TLB and virtual memory hierarchy is used in a asymptotically optimal way, by algorithms designed in the ideal-cache model.



Figure 7.18: TLB misses for priority-queue sort on AMD, zoomed.

7.5 Using 64-bit Computers

In Sections 3.5 on page 59, we described the limitations of 32-bit address space, which imposes a limit on some data structures, like the two cache-oblivious priority queues. We also showed that this problem could be solved by using 64-bit computers with an operating system capable of overcommitting memory.

In order to investigate this further, we managed, shortly before our deadline, to get hold of a Digital Alpha workstation, on which we could install AlphaLinux. This computer has a 64-bit CPU, and the version of Linux to Alpha supports 40-bit addressing, and supports overcommitting. When enabling overcommitting, this, in theory, allows for data structures holding up to 2^{40} bytes, which corresponds to 256 Giga integers. This is, of course, bounded by the actual amount of physical memory and swap space.

The specifications of the computer are:

	Alpha EV56 500Mhz CPU
Brand:	Digital
System:	RedHat Linux 7.2 for Alpha
RAM:	128MB
Cache:	8KB level 1 & 96KB level 2
Swap Space:	8GB

Our purpose of performing the benchmarks on this computer, was to determine how the cache-oblivious algorithms compete on data too large for 32-bit computers, to see how the algorithms perform with very large data sets. We perform the benchmarking with one dataset of 256 Mega integers, which corresponds to 1GB of data, which is larger than the theoretical maximum on the 32-bit benchmark computers, because of the described limitations.

The table in Figure 7.19, shows the results of the benchmarks of the algorithms we managed to run on the Alpha.

	DISTHEAPALTRBSTL	FUNNELHEAPALTSTL	SEQUENCE	STL
insert	194.0	52.6	2.8	4.8
extract	20.5	10.7	2.3	-3
construction	24.0	9.0	3.8	55.2
priority-queue sort	48.5	19.7	6.1	-3

Figure 7.19: Wall-clock time in microseconds per element for 64bit benchmarks.

These results clearly illustrate, that the SEQUENCE implementation still, by far, outperforms the other implementations. STL is able to compete on insertion, but never finishes the extraction of elements. Both of the cache-oblivious implementations have problems performing inserts, which is the operation, in both algorithms, where the major part of the work is performed.

We also tried to benchmark the implementations with the monotone nondecreasing sequence. The benchmarks of SEQUENCE finished in approximately 25 minutes, whereas none of the other finished within 10 hours.

The results from the benchmarking on the 64-bit computer, further support our results, that the cache-oblivious implementations can compete with the RAMmodel implementations, when processing data larger than main memory, but that the cache-aware SEQUENCE implementation is clearly the best.

7.6 Summary

In Section 1.4, we discussed the development in work and I/Os. The conclusion was that although the relative difference between CPU and main memory speeds has increased, the technology used, when making caches, has followed the CPU

development closely. This implies that the gap between CPU and caches are not as big as might be expected, and the advantage of using cache-optimized algorithms for these levels in the memory hierarchy, might not be big enough to justify the extra work. This fact is clearly illustrated in the benchmark results, where none of the cache-oblivious implementations are able to match any of the competitors for small data sets. The cache-aware implementation SEQUENCE, which is tuned to each cache level, is, on the other hand, able to compete with the RAM-based algorithms. Our benchmarks show that this is due to the very small amount of extra work performed by this algorithm. Both the results from the Sparc, as well as the measurements with the large data type, showed the same tendencies as we have described in this chapter.

The cache-oblivious priority queues look very promising and interesting, but our results indicate that the work involved is too large to compete with standard RAM-algorithms, until disk is needed. The amount of extra work imposed by the algorithms, is larger than the gain obtained by efficient cache performance.

Once the datasets are large enough to require the disk, though, the trend is turned and the cache-oblivious priority queues work better than RAM-algorithms. This is because the very high penalty for a page miss, leaves plenty of room for doing extra CPU-work. The overall performance of the different implementations is illustrated by Figure 7.20.



Figure 7.20: Wall-clock time for priority-queue sort on AMD, zoomed.

The cache-oblivious approach is a beautiful model, in theory, but there are a number of interesting problems to solve. The question remains whether it is a viable approach to be used in a wide variety of applications. The key argument for using cache-oblivious algorithms is that they—contrary to the cache-aware implementations—are portable between different architectures, without having to be tuned to the specific memory system. If the need for large data processing is high, it should be possible to spend time tuning a cache-aware algorithm to the current memory system. If the cache-aware algorithms could tune themselves through a series of automated tests, the algorithms could be placed in a system library and the tuning could be performed automatically upon installation.

The problem could be solved with compilers that offer the possibility of obtaining the values of B and M for each of the memory levels on the current computer. This would allow for easy distribution of cache-aware implementations, as the specialization is performed automatically by the compilers. Another approach for programs which are not distributed in source code, could be to run an automatic self-tuning tool, during installation of the software.

For the two algorithms described, the amount of data which needs to be processed, have to be lager than main memory, in order for the cache-oblivious approach to perform well. It should be noted, though, that we have only investigated two cache-oblivious data structures, which both rely on the cache-oblivious technique of using buffers for lazy evaluation. There are other examples of efficient cache-oblivious algorithms, like Prokop's divide-and-conquer implementation of a Jacobi multi pass filter [44].

Chapter 8

Conclusion

Universities are truly storehouses of knowledge: Students arrive from school confident that they know very nearly everything, and they leave years later certain that they know practically nothing. Where did the knowledge go in the meantime? Into the university, of course, where it is carefully dried and stored.

— The Science of Diskworld by Terry Prachett, Ian Steward & Jack Cohen

In this thesis, we set out to investigate the practical value of the theory of cacheoblivious algorithms. We did this by looking into two recently published data structures for priority queues. The principal achievements of our work—in order of appearance—have been:

- I. The cache-oblivious theory was discussed in Chapter 2, with focus on how it relates to real-life computers. We tried to bridge the gap between theory and practice, by describing the differences between the assumptions in the theory and the way contemporary computers are constructed. We found that the tall-cache, automatic replacement, and optimal replacement assumptions are viable abstractions, while the assumption of full associativity is not possible to establish, as the large majority of caches do not have this property. Algorithms that rely on this assumption, might therefore perform arbitrarily worse in practice than suggested by the ideal-cache model.
- II. In Section 2.3.6, we showed that the TLB and the virtual memory system *are* used optimally by cache-oblivious algorithms. The ideal-cache model was not originally described to take this into account. The virtual memory system can, however, be seen as additional levels in the memory hierarchy. Because an optimal algorithm, in the ideal-cache model is optimal on all levels, it is also optimal on the virtual-memory levels.

III. In Chapter 3, we developed solutions for various implementation problems for the two priority queue algorithms. The Distribution Heap was adapted to handle elements with the same priority, and we also described how to insert and extract single elements by the use of small insert and extract buffers. We furthermore clarified how the global rebuilding procedure makes the data structure use linear space—something which was not clearly described in the original article by Arge et al. [6].

For the Funnel Heap, we relaxed the requirement that links had to be allocated in one contiguous chunk of memory in order to make the algorithm use linear space, as was intended, but not described, in the original article by Brodal and Fagerberg.

- IV. We described the problem of limited address space in Section 3.5. This problem limits cache-oblivious data structures, which allocate more space than used for data. For example, the Distribution Heap needs to allocate space for four times the number of elements actually worked on in worst case. If an operating system allows overcomitting the physical memory, the problem is not so serious on 64-bit computers, but this is not necessarily a viable approach.
- V. In Chapter 4, we combined ideas from the Interval Heap [53] and the Distribution heap, to construct a new priority deque, which we named the *Distribution Interval Priority Deque*—or *DIPD*. The DIPD is I/O, work, and space optimal, and to our knowledge the first, cache-oblivious priority deque.
- VI. To examine the practical relevance of cache-oblivious priority queues, we ran thorough benchmarks on real-life hardware. The results of the benchmarks were described in Chapter 7, and all the results are included in Appendix B. Due to ambiguities in one of the articles, we only had time to perform the benchmarks of the Distribution Heap implementations once, as described in Section 6.6. This potentially increases the uncertainty of the measurements, but as none of the other benchmarks which we did run five times displayed any significant variance, it is still possible to make conclusions based on these measurements.

We compared RAM-based, cache-aware, and cache-oblivious algorithms. The results illustrate that the extra work incurred by the cache-oblivious approach is too great for the examined algorithms to be competitive on the smaller levels in the memory hierarchy. Once the data size exceeds the size of main memory, however, the cache-oblivious algorithms are better than RAM-based algorithms, and are, in some situations, able to compete with cache-aware implementations.

Our findings indicated that in the lower levels of cache, the extra work introduced by the—compared to the RAM-model—more complex algorithms, do not make up for the gain achieved by improved use of the small cache levels. The data size needs to be of a size greater than main memory, in order for the cache-oblivious data structures to be faster than standard RAM-model algorithms. Furthermore, it seems that the cache-aware approach is able to offer better overall performance.

We are not able to make conclusions about other cache-oblivious algorithms than those implemented in this thesis, but our research has taken a step towards a comprehensive empirical study of this type of algorithmic design.

As discussed in Chapters 1 and 2, the main incentive for the development of the cache-oblivious theory was the development in CPU and memory speed. Computer hardware will continue to develop, and for the cache-oblivious theory this development is therefore important. Future development could either render the theory, and the algorithms, completely worthless in practice, or, on the other hand, make it an important tool for exploiting the full potential of the hardware. If, for example, large and fast storage media evolve, then the disk and main memory of contemporary computers could melt together and old-fashioned RAM-algorithms would have a revival. On the other hand, the amount of data that must be processed in the future, might grow even more and make efficient use of secondary storage an even bigger issue than it is today. As the future development is not obvious, we must settle with satisfying our curiosity, and try to invent new and better ways to make efficient use of the available memory hierarchies. The cache-oblivious approach seems promising, but it is still quite new and many issues have yet to be examined.

One of the major arguments for using cache-oblivious implementations, instead of cache-aware implementations, is that they can easily be moved between computers, without the need for specialization for each computer. This advantage could, however, also be achieved by cache-aware algorithms, if the feat of obtaining informations about the computer running software could be automated.

The other major advantage advocated by the cache-oblivious theory, is that the algorithms are simpler to design and implement, when they do not have to take multiple memory levels into account. A simple count of the number of lines of source code of our implementations, compared with Sanders cache-aware implementation of the sequence heap—which does take several levels into account—does not support this claim. Our Funnel Heap implementation uses 1300 lines of code, the Distribution Heap 1700 lines, and the Sequences Heap is implemented using 1300 lines of code. Even though the amount of code is not the best measure for complexity, all in all, this does not indicate that cache-aware algorithms should be significantly more complex to implement.

Still, the cache-oblivious theory is beautifully simple, and for many applications it might be the right approach. The only way to find out is to continue the development of more cache-oblivious algorithms and to conduct more experiments to establish the practical usefulness of this design paradigm. An important goal is to look into the constants involved in order to reduce the work overhead if the algorithms are to be competitive on the smaller levels in the memory hierarchy.

8.1 Further Work

There are a number of interesting issues that we have chosen not to pursue within the scope of this thesis. We therefore list a couple of suggestions for possible related future work. Our focus has been on how the theory relates to practice, so these are issues which are interesting in regard to this point of view:

- I. Bilardi et al. [11] showed a separation between cache-oblivious and cache-aware algorithms in the HRAM model. Generalizing/expanding this, by both transferring their results to other memory models¹, and especially transferring this to the ideal cache model would be interesting. Similarly, it would be interesting to try to transfer their results to contemporary computers by showing that there exist computers that have *sufficiently different memory latency functions* to actually achieve the separation between the two types of algorithms in practice.
- II. The ideal–cache model assumes full associativity, which is not realistic in practice. This aspect can simply be ignored for algorithms for which the full associativity is not used in the analysis of the I/O complexity. This is, e.g., the case for all three cache-oblivious algorithms presented in this thesis.

An interesting study would be to find a cache-oblivious algorithm which relies heavily on full associativity, and then examine its behavior in a cache simulator. Here, it will be possible to study the influence of the associativity of the caches, by simulating different associativities.

III. One of the initial motivations for developing the cache-oblivious theory was that the external-memory approach, was thought to have certain drawbacks. The main problem percieved was, that it could be difficult to design and implement algorithms which take several levels of the memory hierachy into account. An interesting study could now be to survey how big this problem is in practice—perhaps by looking at code and interviewing programmers and algorithm designers. Our implementations of cache-oblivious priority queues do not seem significantly less complex than for example Sander's implementation SEQUENCE. Still, SEQUENCE obtains very good performance with a locked set of tuning values, indicating that one set of values *can* work on several architectures. Our examination of priority-queue algorithms has shown that the cache-oblivious approach has a noticable overhead, so it could be interesting to try to combine the best of the two worlds.

¹This is also mentioned by Frigo et al. [22].

Bibliography

There's no quote for the bibliography

- Maz Spork

- A. AGGARWAL, B. ALPERN, A. CHANDRA, AND M. SNIR, A model for hierarchical memory, *Proceedings of the 19th annual ACM conference on Theory of computing*, ACM Press (1987), 305–314.
- [2] A. AGGARWAL AND J. S. VITTER, The input/output complexity of sorting and related problems, *Communications of the ACM* 31,9 (1988), 1116–1127.
- [3] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company (1974).
- [4] L. ARGE, The buffer tree: A new technique for optimal I/O-algorithms, Proceedings of the 4th Workshop on Algorithms and Data Structure, Lecture Notes in Computer Science 955, Springer-Verlag (1995), 334–345.
- [5] L. ARGE, External Memory Data Structures, Handbook on Massive Data Sets, Kluwer Academic Publishers (2001) Chapter 9.
- [6] L. ARGE, M. A. BENDER, E. D. DEMAINE, B. HOLLAND-MINKLEY, AND J. I. MUNRO, Cache-oblivious priority queue and graph algorithm applications, *Proceedings of the 34th ACM Symposium on Theory of Computing*, ACM Press (2002), 268–276.
- [7] L. ARGE, L. TOMA, AND J. S. VITTER, I/O-efficient algorithms for problems on grid-based terrains (extended abstract), *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments, Lecture Notes in Computer Science*, Springer-Verlag (2000), 217–236.
- [8] M. D. ATKINSON, J. R. SACK, N. SANTORO, AND T. STROTHOTTE, Minmax heaps and generalized priority queues, *Communication of the ACM* 29 (1986), 996–1000.

- [9] M. A. BENDER, Z. DUAN, J. IACONO, AND J. WU, A locality-preserving cache-oblivious dynamic dictionary, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM Press (2002), 29–38.
- [10] J. BENTLEY, Programming Pearls, 2nd Edition, Addison-Wesley (2000).
- [11] G. BILARDI AND E. PESERICO, A characterization of temporal locality an its protability across memory hierarchies, *Proceedings of the 28th International Colloquium an Automata, Languages and Programming, Lecture Notes in Computer Science* 2076, Springer-Verlag (2001).
- [12] BOOST, Boost C++ libraries, priority queues library, Worldwide Web Document (2002). Available at www.boost.org/libs/pri_queue/ index.html.
- [13] K. BRENGEL, A. CRAUSER, P. FERRAGINA, AND U. MEYER, An experimental study of priority queues in external memory, *Proceedings of the 3rd Workshop on Algorithm Engineering, Lecture Notes in Computer Science* 1668, Springer-Verlag (1999), 345–360.
- [14] G. S. BRODAL AND R. FAGERBERG, Funnel heap—a cache oblivious priority queue, *Proceedings of the 13th Annual International Symposium on Algorithms and Computation, Lecture Notes in Computer Science* 2380, Springer-Verlag (2002), 426–438.
- [15] G. S. BRODAL, R. FAGERBERG, AND R. JACOB, Cache oblivious search tress via binary tress of small height, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM Press (2002), 39–48.
- [16] S. CARLSSON, Deap A Double-Ended Heap to Implement Double-Ended Priority Queues, Technical report, Lund University (1986).
- [17] S. CARLSSON, The deap—A double-ended heap to implement double-ended priority queues, *Information Processing Letters* 26 (1987), 33–36.
- [18] J. L. CARTER AND M. N. WEGMAN, Universal classes of hash functions, Journal of Computer and System Sciences 18,2 (1979), 143–154.
- [19] T. H. CORMEN, C. STEIN, C. E. LEISERSON, AND R. L. RIVEST, An Introduction to Algorithms, 2nd Edition, The MIT Press (2001).
- [20] E. D. DEMAINE, Preliminary lecture notes: Cache-oblivious algorithms and data structures, *Proceedings of the BRICS Summerschool on Massive Datasets 2002, Lecture Notes in Computer Science*, Springer-Verlag (2002).
- [21] R. W. FLOYD, Permuting information in idealized two-level storage, Proceedings of Symposium on Complexity of Computer Computations, Plenum Press (1972), 105–109.

- [22] M. FRIGO, C. E. LEISERSON, H. PROKOP, AND S. RAMACHANDRAN, Cache-oblivious algorithms (extended abstract), *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press (1999), 285–297.
- [23] FREE SOFTWARE FOUNDATION, Gcc, Website accessible at www.gnu. org/software/gcc/(2002).
- [24] FREE SOFTWARE FOUNDATION, Gnu general public license, Worldwide Web Document (1995). Available at http://www.gnu.org/ copyleft/gpl.html.
- [25] J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture: A Quantitative Approach*, 3th Edition, Morgan Kaufmann Publishers Inc. (2003).
- [26] INTEL CORPORATION, Pentium® pro family developer's manual, volume 2, Worldwide Web Document (1995). Available at http://www.intel. com/design/pro/MANUALS/24269101.pdf.
- [27] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO), ISO/IEC 14882: Standard for the C++ Programming Language, Genevé (1998).
- [28] D. W. JONES, An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM* 29,4 (1986), 300–311.
- [29] J. KATAJAINEN AND J. L. TRÄFF, A meticulous analysis of mergesort programs, *Proceedings of the 3rd Italian Confrence on Algorithms and Complexity, Lecture Notes in Conputer Science* **1203**, Springer-Verlag (1997), 217– 228.
- [30] D. E. KNUTH, *The Art of Computer Programming—Sorting and Searching*, 2nd Edition, Addison-Wesley (1998).
- [31] A. KOENING AND B. E. MOO, Performance: Myths, measurements, and morals, part 6, *Journal of Object Oriented Programming* (2000), 30–33.
- [32] V. KUMAR AND E. J. SCHWABE, Improved algorithms and data structures for solving graph problems in external memory, *In Proceedings of the 8th Symposium on Parallel and Distributed Processing*, IEEE (1996), 169–177.
- [33] A. LAMARCA AND R. E. LADNER, The influence of caches on the performance of heap, *The ACM Journal of Experimental Algorithmics* 1 (1994).
- [34] ALGORITHMIC SOLUTIONS, LEDA, Worldwide Web Document (2002). Available at www.algorithmic-solutions.com/as_html/ products/products.html.

- [35] MAX-PLANCK-INSTITUT FÜR INFORMATIK SAARBRÜCKEN, LEDA-SM, Worldwide Web Document (2002). Available at www.mpi-sb.mpg.de/ ~crauser/leda-sm.html.
- [36] A. M. LIAO, Three priority queue applications revisited, *Algorithmica*,7 (1992), 415–427.
- [37] BITMOVER, Lmbench, Worldwide Web Document (2002). Available at www.bitmover.com/lmbench/.
- [38] L. W. MCVOY AND C. STAELIN, Imbench: Portable tools for performance analysis, *Proceedings of the USENIX Annual Technical Conference*, USENIX (1996), 279–294.
- [39] G. E. MOORE, Cramming more components onto integrated circuits, *Electronics Magazine* 38 (1965), 114–117.
- [40] D. R. MUSSER, Introspective sorting and selection algorithms, *Software -Practice and Experience* **27**,8 (1997), 983–993.
- [41] C. NYBERG, T. BARCLAY, Z. CVETANOVIC, J. GRAY, AND D. LOMET, Alphasort: A cache-sensitive parallel external sort, *VLDB Journal* 4 (1995), 603–627.
- [42] J. H. OLSEN AND S. SKOV, A comparative analysis of three different priority deques, CPH STL Report 2001-14, Department of Computer Science, University of Copenhagen (2001). Available at http://www.cphstl.dk.
- [43] UNIVERSITY OF TENNESSEE, Performance application programming interface library, Worldwide Web Document (2002). Available at http://icl. cs.utk.edu/projects/papi/.
- [44] H. PROKOP, Cache-oblivious algorithms, M. Sc. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1999).
- [45] N. RAHMAN, R. COLE, AND R. RAMAN, Optimised predecessor data structures for internal memory, *Proceedings of the 5th International Workshop on Algorithmic Engineering*, *Lecture Notes in Computer Science* 2141, Springer-Verlag (2001), 67–78.
- [46] P. SANDERS, Fast priority queues for cached memory, *The ACM Journal of Experimental Algorithmics* **5** (2000).
- [47] J. F. SIBEYN, External selection, Lecture Notes in Computer Science 1563 (1999), 291–301.

- [48] SILICON GRAPHICS COMPUTER SYSTEMS, Standard template library programmer's guide, Worldwide Web Document (2001). Available at www. sgi.com/Technology/STL/.
- [49] SILICON GRAPHICS COMPUTER SYSTEMS, INC., Priority_queue<t, sequence, compare>, Worldwide Web Document (2001). Available at www. sgi.com/tech/stl/priority_queue.html.
- [50] D. D. SLEATOR AND R. E. TARJAN, Amortized efficiency of list update and paging rules, *Communications of the ACM* **28**,2 (1985), 202–208.
- [51] M. SPORK, Design and analysis of cache-conscious programs, M.Sc. Thesis, Department of Computer Science, University of Copenhagen (1999). available at: www.diku.dk/forskning/ performance-engineering/Publications/spork99.pdf.
- [52] DUKE UNIVERSITY, TPIE, a transparant parallel I/O environment, Worldwide Web Document (2002). Available at www.cs.duke.edu/TPIE/.
- [53] J. VAN LEEUWEN AND D. WOOD, Interval heaps, *The Computer Journal* **36**,3 (1993), 209–216.
- [54] J. S. VITTER, External memory algorithms and data structures: Dealing with massive data, ACM Computing Surveys 33,2 (2001), 209–271.
- [55] J. S. VITTER AND E. A. M. SHRIVE, Algorithms for parallel memory I: Two-level memories, *Algorithmica* 13 (1994), 110–147.
- [56] L. YDE, *Performance Engineering the nth_element Function*, CPH STL Report 2002–04, Department of Computing, University of Copenhagen (2002).

List of Figures

1.1	A typical memory hierarchy in contemporary computers 3
1.2	A memory hierarchy with a virtual memory system 6
1.3	Table of latencies of cache and memory 9
2.1	The ideal-cache model
2.2	Associativity and cache sizes on real hardware
2.3	The multilevel ideal cache model
2.4	Abstraction of the virtual memory system
2.5	The recursive van Emde Boas layout. 26
3.1	The layout of a Distribution Heap [6]
3.2	Example illustrating a Distribution Heap
3.3	The structure of down buffers
3.4	Split with equal elements
3.5	The structure of a funnel
3.6	Illustration of the structure of a Funnel Heap [14]
3.7	Illustration of the memory layout of a Funnel Heap
3.8	Example illustrating a Funnel Heap
3.9	Illustration of a <i>sweep</i> in the Funnel Heap
4.1	Illustration of the MinMax priority deque
4.2	Illustration of the Deap
4.3	Illustration of the Interval Heap
4.4	Illustration of the DIPD
4.5	Illustration of the intervals in the priority deque
5.1	Table of PAPI events for level 1 cache. 77
5.2	Table of PAPI-events we used in the benchmarks. 80
6.1	Summary of the sequences used
6.2	Table of short names in implementation 91
7.1	Wall-clock time for inserts on AMD
7.2	Wall-clock time for inserts on Sparc
7.3	Wall-clock time for construction on AMD

7.4	Level 1 cache misses for extracts on AMD	96
7.5	Level 1 cache accesses for the complex sequence on AMD, zoomed.	97
7.6	Level 1 cache misses for the complex sequence on AMD	97
7.7	Level 2 cache misses for the complex sequence on AMD	98
7.8	Wall-clock time for the complex sequence on AMD	99
7.9	Wall-clock time; complex sequence with large data type	99
7.10	Wall-clock time for monotonic sequence on AMD	100
7.11	Wall-clock time for priority-queue sort on AMD, zoomed	100
7.12	Wall-clock time for extracts on AMD	101
7.13	Page Faults for extracts on AMD	102
7.14	Page Faults for inserts on AMD	102
7.15	Page Faults for inserts on AMD, zoomed.	103
7.16	TLB misses for monotonic sequence on AMD.	105
7.17	TLB misses for priority-queue sort on AMD.	106
7.18	TLB misses for priority-queue sort on AMD, zoomed	107
7.19	Wall-clock time for 64-bit benchmarks.	108
7.20	Wall-clock time for priority-queue sort on AMD, zoomed	109

List of Algorithms

1	Distribution Heap: insert	45
2	Distribution Heap: push to level $X \ldots \ldots \ldots \ldots \ldots \ldots$	46
3	Distribution Heap: split	47
4	Distribution Heap: extract	47
5	Distribution Heap: pull from level $X \ldots \ldots \ldots \ldots \ldots$	47
6	Funnel Heap: extract	52
7	Funnel Heap: insert	52
8	Funnel Heap: sweep	54
9	Distribution Heap: construction	57
10	Funnel Heap: construction	58
11	DIPD: push to level X	68
12	DIPD: distribute_max to level X	69
13	DIPD: pull_min from level $X \ldots \ldots \ldots \ldots \ldots \ldots \ldots$	70
14	DIPD: insert	71
15	DIPD: extract_min	72
16	DIPD: construction	73

Symbols and Abbreviations

1st level of cache	2
2nd level of cache	2
Least Recently Used. A replacement policy	4
Translation Lookaside Buffer. A cache used to cache transla-	
tions from virtual to physical address	7
One load or store of data to a memory or cache. Is also gener-	
ally used as a unit of measure, to denote the number of memory	
accesses	13
The size of the problem currently worked on	13
The number of elements that fit into internal memory	13
The number of elements that fit into one block	13
The I/O cost of scanning N elements. Equal to $O\left(\frac{N}{B}\right)$	14
The asymptotic optimal number of I/Os used for sorting N ele-	
ments. Equal to $O\left(\frac{N}{B}\log_{M/B}\left(\frac{N}{B}\right)\right)$	14
An abbreviation of <i>Distribution Interval Priority Deque</i>	64
	1st level of cache

Appendix A

Source Code

This software comes with ABSOLUTELY NO WARRANTY. Even if it erases your hard drive, too bad. Although we did fix that bug from the last release.

- README from a long-ago release of DJGPP

To save a small forest, we have chosen to enclose source code, benchmark system and results of our performance measurements in digital form. For the official versions of this thesis, we enclose the package of files on a supplemental CD-ROM. For other readers, it is possible to download the files from:

www.dunkel.dk/thesis

Included here are the source code for our implementations, and the benchmark results. The source code is found in the directory src/ and is enclosed in two ways. First, we have made a reader-friendly version, where the files are typeset and collected in one document. The file code.ps is a Postscript version of this, and code.pdf is the same document in PDF format.

Second, the individual files are included, to allow for other to use our implementation or part of it for further work. We enclose a short description of each of the files, and which page they are found on in the source code collections.

The directory src/ contain the following subdirectories, and files:

File name	Description	Page	
The directory heap	/distHeap/src/ containing:		
distHeap.h	The interface to the Distribution Heap.	1	
distHeap.cpp	Implementation of the Distribution Heap operations.	2	
level.h	Implementation of a level, used by distHeap.cpp.	8	
The directory heap	/funnelHeap/src/ containing:		
funnelHeap.h	Interface and implementation of the Funnel Heap.	15	
link.h	Implementation of the link-class.	19	
k_way_merger.h	Implementation of a K-way merger.	20	
bin_merge.h	Implementation of a binary merger.	22	
buf.h	Implementation of circular buffer.	24	
funnelSort.h	Implementation of Funnel Sort, this is used for ver-	26	
	sions of both Distribution Heap and Funnel Heap		
	that uses a cache-oblivious optimal sorting method.		
The directory heap	/ containing:		
heapTest.h	The general priority queue benchmark system, that	27	
	runs the benchmarks.		
reverse_comp.h	trick implementation that can reverse a Strict Weak	30	
	Ordering, used to reverse the comparison function		
	passed to sort.		
largeType.h	The large data type, used for benchmarking.	31	
The directory heap	/empty containing:		
empty.h	The dummy implementation with empty functions	32	
	used to determine the overhead in the benchmarking		
	system.		
The directory heap	/testAllocate containing:		
alloc.cpp	Implementation used to simulate Distribution Heap	33	
	allocation. Used when trying to find the best value		
	of c in the Distribution Heap.		
The directory template / containing:			
main.h	Implementation of our general benchmark system,	34	
	that can be used for performing various benchmarks		
	on implementations.		
out_of_store.h	A function to ensure an informative error message	36	
	when an implementation runs out of address space.		

Appendix B

Benchmark Results

Real programmers don't comment their code. It was hard to write, it should be hard to understand.

- Unknown

We have also included graphs showing the results of all the benchmarks we have performed—both on the web page and CD. The benchmark results are located in the directory benchmarks/plots/. For the analysis of the Level 1 and 2 caches, it is interesting to "zoom" in on the behavior in these by restricting the range of x-values shown. Graphs showing the same benchmarks, but zoomed for this are stored in the directory L1L2/. On some of the benchmarks, the difference between the worst and the best implementations are so large that it is not possible to get an precise reading of both, when they are shown in the same graph. For these, we have furthermore made a graph showing the best implementation. This is done by limiting the y-values shown. These graphs are found in the directory Yzoom/. This implies that all benchmarks are shown in two or three graphs each.

The filenames of the graphs describes the type of benchmark they represents. All files are found in both Postscript and PDF format. The structure of the filenames are: <arch>-<measure>-<sequence>.(ps|pdf)

The interpretation of the fields in the filenames are:

- **arch** The architecture the benchmark was performed on. Linux is the AMD, and SunOS in the Sparc.
- **measure** What was measured. timeit is wall-clock time and proc is number of major page faults. Further more we have—as described in Section 5.2.2 on page 76—performed a number of performance measurements using PAPI, for these measure is the name of the PAPI event. The measurements with the large data type are furthermore denoted with "Large".
- sequence The operation sequence that was measured. i is insert, e is extract, u is construct, ue is simulation of priority queue sort, m is monotonic nondecreasing data sequence, and k is long operation sequence.

We also include the results of the profiling of our implementations. These are found in the directory benchmarks/profiles/. The filenames are structured as:

```
<implementation>-prof-<sequence>.txt,
```

where implementation is the short name of the implementation, and sequence is the sequence of operations, with the same abbreviations as above.