Cache-Oblivious Searching and Sorting

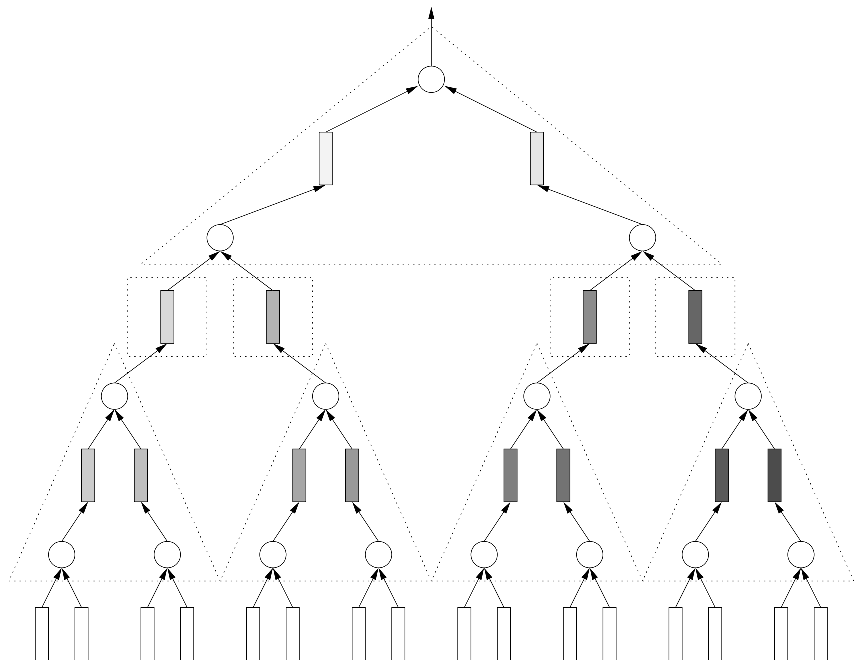
Master's Thesis

By

Frederik Rønn
frederik@diku.dk

Department of Computer Science
University of Copenhagen

July 1, 2003



Abstract

Algorithms that use multi-layered memory hierarchies efficiently have traditionally relied on detailed knowledge of the characteristics of memory systems. The cache-oblivious approach changed this in 1999 by making it possible to design memory-efficient algorithms for hierarchical memory systems without such detailed knowledge. As a consequence, one single implementation of a cache-oblivious algorithm is efficient on any memory hierarchy. The purpose of the thesis is to investigate the behavior of cache-oblivious searching and sorting algorithms through constant-factors analysis and benchmarking.

Cache-oblivious algorithms are analyzed in the ideal-cache model, which is an abstraction of real memory systems. We investigate the assumptions of the model in order to determine the accuracy of cache-complexity bounds derived by use of the model.

We derive the constant factors of the cache complexities of cache-oblivious, cache-aware, and traditional searching and sorting algorithms in the ideal-cache model. The constant factors of the work complexities of the algorithms are derived in the pure-C cost model. The analyses are verified through benchmarking of implementations of all algorithms.

For the searching algorithms, our constant-factors analysis predicts the benchmark results quite precisely — considering both memory performance and work complexity. For the more complex sorting algorithms our results show the same pattern, though the similarities between predicted and measured performance are not as significant.

Furthermore, we develop a new algorithm that lays out a cache-oblivious static search tree in memory in linear time, which is an improvement of the algorithms known so far.

We conclude that by combining the ideal-cache model and the pure-C model, the relative performance of programs can be predicted quite precisely, provided that the analysis is carefully done.

Preface

This thesis is submitted as partial fulfillment of the Danish Kandidatgrad i Datalogi (*condidatus scientiarium in Computer Science*) at the University of Copenhagen. The thesis was written under the supervision of Jyrki Katajainen. It is expected that the reader has an understanding of computer science corresponding to that of a graduate level student. No prior knowledge of analyzing I/O or cache complexity is expected.

Overview of the Thesis

The topic of this thesis is cache-oblivious searching and sorting. Our main goal is to investigate whether it is possible to predict the behavior of these algorithms by use of constant-factors analysis of the work and cache complexities of these algorithms.

We analyze cache-oblivious, cache-aware, and traditional searching and sorting algorithms in detail and verify the validity of these theoretical results by benchmarking implementations of all algorithms of concern. The thesis is organized as follows:

- In **Chapter 1** we explain why it is of importance to take the memory system into account when designing algorithms. We briefly introduce the notion of cache-obliviousness and takes a tour of the practical work done so far on cache-oblivious algorithms. Based on this tour we explain why it is interesting to analyze the constant factors of these algorithms.

- Knowing contemporary memory systems and the current trends in their design is important for understanding the cache-oblivious approach and the ideal-cache model in which cache-oblivious algorithms are analyzed and designed. In **Chapter 2** we review the aspects of modern memory systems that are the most important in relation to cache-oblivious algorithms. Furthermore, the impact of memory latency on algorithm design is illustrated by use of a simple example.

- In **Chapter 3** we present the ideal-cache model and carefully investigate the ways in which the model differs from real memory systems. This is important in order to understand how to interpret the constants in cache complexity bounds derived in the model. Furthermore, the chapter includes a description of the related external-memory model as well as a description of the hierarchical-memory model. The latter description is included to illustrate another way of modeling multi-layered memory systems.

- In **Chapter 4** we review two computational models, namely the MMIX-model and the pure-C model. Both of these models can be used for meticulous work complexity analysis of algorithms in general. Based on this review we decide on the model to use for our analysis.

- With our apparatus of analysis presented, we turn to analyzing static search trees in **Chapter 5**. We carefully analyze the cache-oblivious static search tree as well as cache-aware and more traditional static search tree variants. The analysis reveals the constant factors of both the work and cache complexity of the various approaches.

  The cache-oblivious search tree uses a special way of laying out data in memory. We present a new algorithm for laying out data in this way.

- The topic of **Chapter 6** is the cache-oblivious sorting algorithm called funnelsort. The algorithm is a variant of mergesort and uses a data structure called a k-funnel to merge elements cache-obliviously. We analyze the constant factors of the cache and work complexity of both the k-funnel and funnelsort.

  In addition, the chapter includes the cache-complexity analysis of another mergesort variant which we compare to funnelsort.

- We present the results of our benchmarks in **Chapter 7**. We comment on the experimental results and compare them to the analytically derived complexity bounds of Chapter 5 and 6. The method used in the performance investigation is described prior to the presentation of the benchmark results.

- In **Chapter 8** we summarize the experimental results and emphasize the contribution of this thesis. We also give some directions of further work.

We have chosen only to include the most relevant source code and benchmark results in the thesis. For the complete source code and benchmark results we refer to the web-site:
`http://www.diku.dk/forskning/performance-engineering/frederik/`.

Acknowledgements

Jyrki Katajainen deserves great thanks for insightful discussions and great feedback throughout the working process — also regarding matters such as working and writing techniques. Thanks to his positive mind, I have always left his office with an optimistic feeling and renewed energy.

Also, I would like to thank the proof-readers Kasper Thiberg, Rudy Negenborn, and Thomas Wang for their efforts, and the inmates of the DIKU penthouse for making it a great place to write a thesis.

Last but not least, I thank Nina for her love and support — especially during the final months of this project.

Frederik Rønn
Copenhagen, July 2003

Contents

CHAPTER 1

Introduction

“Begin at the beginning,” the King said, gravely, ”and go on till you come to the end; then stop.”
— Lewis Carroll, Alice in Wonderland

When we analyze an algorithm our aim is to predict the resources that the algorithm requires. Usually we are interested in the computational time of the algorithm or the amount memory it uses. In other words, we analyze the *work complexity* and the *space complexity* of an algorithm.

Algorithm analysis is usually a purely analytical task where we describe the resource usage independent of specific computer architectures and models. Therefore, algorithms are analyzed in *computational models*, that is, abstractions of how real computers work. It is important that these models are both simple, so that the analysis task is relatively easy, and sufficiently detailed, so that the reliability of the analysis is ensured.

When we consider the impact that the memory system of a computer has on the running time of programs, it is most often the latency associated with transporting data from and to the storage system, e.g. magnetic disks, that is the main objective.

The interest in minimizing the amount of disk accesses is as important as it is obvious. Accessing data on disk may take several milliseconds and only a few hundred megabytes of data can be transferred from disk to main memory per second. In comparison, the access time of main memory is 80 – 250 nanoseconds and data can be transferred at a speed of several gigabytes per second [21].

However, in spite of the fact that main memory is faster than disks, main memory accesses are becoming increasingly more influential on the running time of programs. For the past many years the annual increase in CPU speed has followed Moore's Law [21], which states that the number of transistors that can be contained on a single chip increases by 55% every year. The annual performance improvement in DRAM[1] latency, is only about 7%. As shown in Figure 1.1 this CPU-DRAM performance gap has widened continuously since 1980[2].

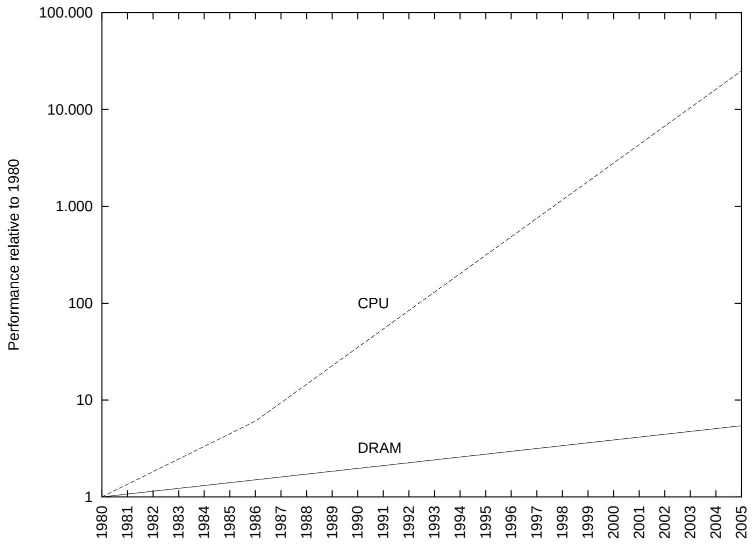


Figure 1.1: The CPU-DRAM performance gap since 1980 with the performance of 1980 as baseline. The improvement in CPU performance was 35% per year before 1986 and 55% thereafter. The annual performance improvement of DRAM latency has been 7%.

Figure 1.1 simplifies the actual situation, since it reveals only one aspect of the performance gap, namely sheer speed. A more realistic picture involves details of the memory hierarchy and instruction execution as well as more general knowledge of operating systems. At the software level, the technique of context switching found in multitasking operating systems provides an example of such a detail. These operating systems decrease the impact memory stalls have on instruction throughput by letting the CPU work on process B while process A waits for the memory system. This clearly optimizes the overall performance of the system, but looking at process A in isolation, the technique of context switching actually increases the

[1]DRAM or *dynamic random access memory* is the technology of choice for main memory.

[2]The figure is a reproduction of Figure 5.2 in [21] and used with permission from Elsevier Science.

running time of that process. Another factor is the ability of some CPUs to recognize and reorganize independent instructions to minimize idle time, while waiting for the memory system.

Due to factors such as these, it is hard to determine whether the performance gap in practice increases as rapidly as depicted by Figure 1.1. It is a difficult task to take into account every aspect of the gap in order to make the analysis precise.

What is worth noting is that the performance gap exists and memory latency therefore is an important factor when analyzing the running time of programs.

As mentioned earlier, computational models are used when analyzing algorithms. The most commonly used model is the *random-access machine* model (RAM-model) [16] — a model of computer machine languages. The RAM-model assumes a flat random-access memory of unlimited size and a uniform access cost to all memory locations. This way of modeling the memory system makes the model unable to capture the increasing impact that memory systems have on the behavior of computer programs.

For more precise predictions of running times, we need computational models that resemble contemporary memory systems more closely. Unfortunately, since modern memory systems are complex, so are the models that represent them. Until recently, models of the memory system have relied on quite detailed knowledge of memory system characteristics, such as memory access times and the sizes of the different memory layers, in order to make good predictions of running time. Therefore, analyzing algorithms in these models have not been as straight-forward as one could have hoped.

1.1 Cache-Oblivious Algorithms

Taking account of memory latency when designing and analyzing algorithms is not a new idea. As mentioned, focus has earlier mainly been on the rather heavy latencies associated with reading data from and writing data to disks and tapes, but in 1987 Aggarwal et al. [1] presented the *hierarchical memory model*. The hierarchical memory model can be used to analyze algorithms in memory hierarchies of multiple layers, but the accuracy of the model's predictions depends on detailed knowledge of the hierarchy itself. In other words, an algorithm that is designed in this model is locked to the specific memory hierarchy in which it was originally designed.

In 1999 Frigo et al. [19] presented a new model of the memory system, the *ideal-cache model*. With this model it became possible to design and analyze algorithms that used the memory system efficiently, but at the same time were unaware of the characteristics of the memory system: The notion of *cache-oblivious algorithms* was introduced[3].

[3]The *ideal-cache model* and the notion of *cache-obliviousness* were first introduced by

The number of cache misses that a cache-oblivious algorithm causes in the ideal-cache model is defined by the algorithm's *cache complexity*. Frigo et al. showed that a cache-oblivious algorithm of asymptotically optimal cache-complexity in the ideal-cache model exhibits asymptotically optimal cache-complexity in any modern memory system. This quality makes cache-oblivious algorithms interesting, since one single implementation of a cache-oblivious algorithm will immediately work well on all computers. Furthermore, cache-oblivious algorithms are appealing since their machine independency potentially make them more elegant than their *cache-aware* counterparts.

1.2 Previous Work on Cache-Obliviousness

Practical work on cache-obliviousness so far has mainly focused on comparing the performance of cache-oblivious implementations to implementations of cache-aware and more traditional[4] algorithms through benchmarking. The limited work done so far indicates that cache-oblivious algorithms can indeed compete with traditional algorithms that do not take the memory hierarchy into account, but are inferior to cache-aware algorithms.

1.2.1 Cache-Obliviousness vs. Traditional Approaches

Frigo et al. showed that cache-oblivious matrix transposition executes in 70% less time than a traditional iterative approach and that cache-oblivious matrix multiplication is almost twice as fast as a traditional implementation involving 3 nested loops. Interestingly, despite the fact that the transpose problem exhibits no temporal locality in its memory reference pattern, the cache-oblivious approach pays off. Prokop [38] found that cache-oblivious Jacobi multipass filters are almost twice as fast as a traditional implementation when the data does not fit in the level 2 cache.

Olsen & Skov [35] showed that two rather complex cache-oblivious priority queues, of optimal work and cache complexity [4, 13], are competitive to traditional implementations only when the data size exceeds the size of main memory. For small input sizes the workarounds to make the algorithms cache-oblivious are too complex to make up for the gains due to the better memory usage. Brodal et al. [15] noticed similar behavior with cache-oblivious search trees. They implemented search trees of various memory layouts and compared their performance. The cache-oblivious layout was

Harald Prokop in his Master's Thesis in 1999 [38] and later the same year published with co-authors Frigo, Leiserson and Ramachandran as an extended abstract [19]. Throughout this thesis we will use the extended abstract of Frigo et al. as our primary source on cache-obliviousness, though many of the same results are described in Prokop's thesis.

[4]By categorizing algorithms as *traditional* we mean algorithms designed to work well in the RAM model or similar models that do not take the memory hierarchy into account.

best for all but small data sizes when compared to the traditional breath-first and depth-first layouts.

1.2.2 Cache-Obliviousness vs. Cache-Awareness

Intuitively, cache-oblivious algorithms should not be able to run faster than their cache-aware competitors, but, as Frigo et al. noted, a gap in asymptotic complexity between cache-aware and cache-oblivious algorithms has not been proved. However, according to recent work of Brodal & Fagerberg [14] optimal comparison-based sorting is not possible by the cache-oblivious approach without the so-called tall-cache assumption[5].

Benchmarks of both Olsen & Skov and Brodal et al. indicated that the running times of cache-oblivious algorithms are comparable to those of cache-aware counterparts.

1.2.3 Theoretical Results

In addition to the cache-oblivious algorithms that have been implemented and benchmarked, a number of algorithms of optimal work and cache complexity has been described and analyzed in the ideal-cache model. Among the most interesting results are those of Frigo et al., who, besides the above-mentioned matrix problems, present two sorting algorithms and an algorithm for fast Fourier transformations. Bender et al. [7] introduced cache-oblivious B-trees, and in a recent paper Arge et al. [4] present a number of graph algorithms utilizing a cache-oblivious priority queue.

On the boundary between theoretical and practical studies on cache-obliviousness is the work of Bender et al. [8]. They studied the behavior of a cache-oblivious dynamic dictionary[6] in a simulated memory hierarchy and compared it to a standard B-tree. Their main objective was to understand how the performance of the dictionary was affected by block size and memory size. Their results indicate that the worst-case performance of the dictionary is at least as good as the worst-case performance of the standard B-tree for typical block and memory sizes.

1.2.4 Evaluating the Ideal Cache

In practice, only few efforts have been made to evaluate the practical relevance of the assumptions of the ideal-cache model. Olsen & Skov [35] investigated the ideal-cache model and made the effort of bridging the gap between the model and actual memory systems of contemporary computers by relating the assumptions of the model to the characteristics of real memory systems. They argue that all the assumptions of the model are viable

[5]We will explain this assumption is Section 3.4.

[6]A simplified version of the cache-oblivious B-tree [7].

abstractions — except for the assumption of full associativity, as the vast majority of modern caches do not have this property.

1.3 Analyzing the Constant Factors

Based on the previous work on cache-obliviousness, there is a number of reasons for investigating the constant factors of the work and cache complexity bounds of cache-oblivious algorithms.

The experimental work on cache-oblivious priority queues of Olsen & Skov indicated that the workarounds to make an algorithm oblivious of the memory system may cause the work complexity of the algorithm to increase by a constant factor. It is interesting to investigate whether this observation holds for other algorithms as well.

Analyzing the constant factors of the work complexity by scrutinizing cache-oblivious algorithms may also reveal ways of optimizing the algorithms and making the cache-oblivious approach advantageous — even in the faster memory layers.

For algorithms of similar asymptotic cache complexity it is interesting to see if constant-factor analysis can predict their relative performance. By combining analyses of work and cache complexities we may be able to predict the behavior of algorithms more closely than by use of only a single metric.

Finally, to our knowledge, no one has earlier investigated the validity of constant factors derived in the ideal-cache model. A fact that in itself makes the topic interesting.

CHAPTER 2

Memory Systems and Latency

"Nothing is more responsible for the good old days than a bad memory."
— Franklin Pierce Adams

Cache-oblivious algorithms are analyzed in the ideal-cache model, and this model is in fact an abstraction of real memory hierarchies. Understanding modern memory systems is therefore essential for understanding the cache-oblivious approach.

In Section 2.1 we explain those aspects of modern memory systems that are the most important in relation to cache-oblivious algorithms. In Section 2.2 we investigate the concept of memory latency, and in Section 2.3 we illustrate the impact of memory latency on algorithm design by use of an example.

2.1 Important Aspects of the Memory Hierarchy

In this section we review the aspects of modern memory systems that are important in the context of the cache-oblivious algorithms. The emphasis is therefore not on describing the details of some vendor-specific system, but rather on presenting the key ideas of some general and widely used concepts. For a more complete presentation refer to [37].

A memory system contains both the data that a program manipulates (i.e., input and output data) and the program code itself. In the context of cache-oblivious algorithms our interest is not in knowing how memory systems store program code. Therefore memory system concepts that are only relevant in that respect are intentionally left out in this chapter.

Modern memory systems are composed of multiple levels of memory placed in a hierarchical structure. A typical memory hierarchy is shown in Figure 2.1. A small but fast memory layer is placed closest to the CPU, and one or more increasingly larger, but slower, memory layers are placed further away from the CPU. Considering the registers as an integrated part of the CPU, a typical hierarchy consists of one or more layers of memory between the CPU and main memory and, as the largest layer, a disk. The term *cache* is used to describe the memory layers between CPU and main memory, so the hierarchy on Figure 2.1 has two cache layers and a disk, for a total of four memory layers. The term cache is also used to describe the role of the smaller of two consecutive memory layers, e.g., the main memory acts as a cache for the disk. Unless otherwise noted, when we use the term cache, it is used in the former meaning.

A distinction is made between the *volatile* part of the memory hierarchy where data disappear when the power is turned off, i.e., the layers closer to the CPU than the disk, and the *non-volatile* part where data remain unchanged after a reboot, i.e., the disk. The volatile layers represent *primary memory* and the disk the *secondary memory*. Sometimes primary and secondary memory are called internal and external memory, respectively.

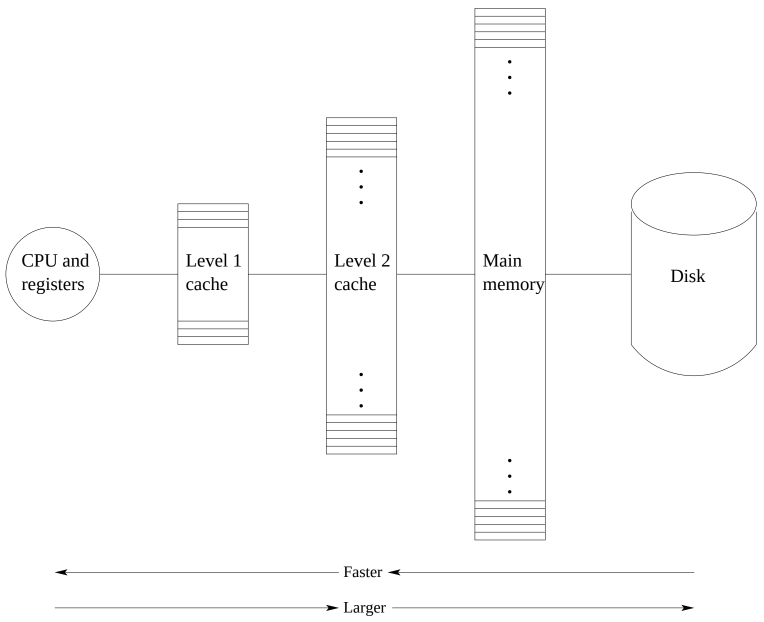


Figure 2.1: The memory hierarchy consists of multiple memory layers of increasing size and decreasing speed. A memory hierarchy may have more levels of cache than depicted.

Ideally, the memory hierarchy adheres to the *inclusion property*, stating that a memory layer closer to the CPU contains a proper subset of the data at any level further away. As such, if the layers are numbered in order of increasing size, then layer i acts as a buffer for layer $i+1$. The layer furthest away from the CPU contains all data. In Figure 2.2 the level 1 and 2 cache characteristics of the Intel® Pentium® family are shown.

Computer	Description
Intel Pentium	8 or 16 KB 2 or 4-way L1 cache, 32-byte cache lines L2 cache is external to CPU and depends on motherboard.
Intel Pentium 2	16 KB 4-way L1 cache, 32-byte cache lines. 256 or 512 KB 4-way L2 cache, 32-byte cache lines.
Intel Pentium 3	16 KB 4-way L1 cache, 32-byte cache lines. 256 or 512 KB 4 or 8-way L2 cache, 32-byte cache lines.
Intel Pentium 4	8 KB 4-way L1 cache, 64-byte cache lines. 128, 256, or 512 KB 2, 4, or 8-way L2 cache, 64-byte cache lines.

Figure 2.2: The cache characteristics of the Intel® Pentium® family. From http://www.sandpile.org.

2.1.1 Principle of Locality of Reference

The memory hierarchy is based on two observations regarding the behavior of computer programs. These observations make up the *principle of locality of reference*:

Temporal locality is the observation that accessing the same memory location is often done many times within a short interval of time. An example is the counter in a `for`-loop, that is both read and written at least once during each iteration. In other words the observation states that, if a memory location has been accessed, then the probability that it will be accessed again in near future is high.

Spatial locality is the observation that accessing memory locations close to one that has recently been accessed are likely to occur. A common programming construct that illustrates this observation is a `for`-loop iterating through an array of elements.

These two principles are very important in relation to cache-oblivious algorithms, since algorithms that are designed with these principles in mind inherently behave memory efficiently.

How the memory hierarchy ensures the principle of locality should become apparent when we describe how and when data are transferred between memory layers.

2.1.2 Moving and Addressing Data in the Hierarchy

The CPU can only access data residing in the closest memory layer, so temporal locality is preserved by keeping data often accessed as close to the CPU as possible.

In case the CPU requests data that are stored in the closest cache layer, the CPU has immediate access to the data and a *cache hit* has occurred. If the closest cache layer does not contain the requested data, then a *cache miss* occurs. Then data have to be transferred from a memory layer further away to the closest cache prior to being accessible by the CPU. In principle, a request from the CPU can therefore cause misses in all but the layer furthest away. Obviously, cache misses do not occur in the layer furthest away because it contains all data.

At any layer of memory the contents are divided into consecutive sequential chunks. In the cache layers, these chunks are called *blocks* or *cache blocks* and may vary in size among different cache layers. Most often a block contains several words[1].

In case of a cache miss spatial locality is preserved by moving data between layers in blocks. Hence, the *block size* at a given cache layer determines the granularity of memory addressing at that particular layer. Memory addressing within the memory hierarchy is therefore different from that of the CPU where memory is addressed in words.

As the memory layers are placed further away from the CPU they get larger, and so does the chunks in which memory is addressed and transferred. At the main memory layer the chunks are often several kilobytes and the term *pages* is used instead of blocks. Consequently, failing to find data in main memory causes a *page fault* instead of a cache miss. The page containing the requested data is then copied from disk to main memory.

The *capacity* of a memory layer is determined by the number of blocks or pages it can contain. Caches are divided into *cache lines* that can contain one block each, so the capacity of a cache is calculated by simply multiplying the block size and the number of cache lines. Main memory is divided into *frames* that can contain one page each. The size of main memory is defined by the number of pages it can contain.

Which block to replace in case of a cache miss is determined by the *associativity* and the *replacement policy* of the cache layer in which the cache miss occurs.

[1]On a 32-bit computer the size of a memory cell is 32 bits. A memory cell can therefore take on any value that can be expressed in 32 bits, e.g., the integers in the range $\{0, \ldots, 2^{32} - 1\}$. In this context a word is the same as a memory cell.

The *associativity* restricts the number of cache lines in which a given block can reside. We call the cache lines that potentially can contain a particular block of memory the *candidate lines* for that block. In other words, the block maps to a number of candidate lines. Caches are divided into three categories based on associativity:

A *fully-associative* cache poses no constraints on where to place a block. The number of candidate lines for any block is the same as the total number of cache lines in the cache, i.e., any block can be placed anywhere.

A *direct-mapped* cache is the most restrictive type since a block has only one candidate line. The index of the candidate line in the cache is calculated as *block address* modulo *number of cache lines.*

A *set-associative* cache is a hybrid of the fully-associative and the direct-mapped caches. An x-way set-associative cache is divided into sets each containing x cache lines. A block of memory maps to exactly one of these sets, but within this set the block can be placed in any line. If the memory is divided into blocks numbered by increasing addresses from 0 to m, then the index of the set in the cache that block a $(0 \leq a \leq m)$ maps to is calculated as $a \bmod x$. The number x defines the *degree of associativity* of the cache. Typically, set-associative caches have degree 2, 4, or 8.

In case of cache misses in set-associative or fully-associative caches some strategy is needed to choose the cache line among multiple candidate lines in which to place the new block. If there are empty lines among the candidates, then the choice is easy, but if all candidate lines contain data, then the choice of which cache line to replace is made according to the *replacement policy* of the cache. To preserve the principle of temporal locality an optimal replacement policy would replace the line that is referenced furthest in the future. Determining what line that would be may involve considerable knowledge of the future sequence of instructions executed by the CPU, so much more simple policies are used in practice.

For 2-way set-associative caches the well-known *least-recently-used* (LRU) policy is implemented by keeping just a single bit of information for each cache line: When a cache line is accessed its *recency bit* is set and the recency bit of its fellow cache line is reset. Theoretically, the LRU policy could be implemented for higher degrees of associativity using more recency bits, but in practice some approximation to LRU, such as the *not-recently-used* policy is used. Even a policy choosing the candidate line to replace at random is used. The point is, that for caches of limited associativity the replacement policy has almost no impact on the number of cache misses. In fact, in 2, 4, and 8-way set-associative caches, choosing the line to replace by random has been observed to work almost as good as LRU replacement in caches containing 16, 64 and 256 KB data in 64-byte blocks [37]. The bigger the cache, the smaller the difference, and for the 256 KB caches the two replacement schemes were equally good. For the smaller cache sizes a higher degree of associativity resulted in fewer cache misses but for the 256 KB caches this tendency was not present.

Deciding on which page to replace in main memory in case of a page fault is of greater importance. To save the penalty of just a few extra page faults makes it worthwhile to use a clever replacement policy in main memory. At the main memory layer, the replacement policy is implemented in software and therefore depends on the operating system.

2.1.3 Categorizing Cache Misses

Cache misses can be divided into three categories depending on the context in which they occur:

Compulsory misses occur when a block is referenced for the first time, i.e., it has not been in the cache earlier. Compulsory misses are therefore unavoidable and occur at every level in the memory hierarchy.

Capacity misses happen when the cache has no empty cache line in which a block that has previously been in the cache can be placed. The size of the cache as well as the quality of the replacement policy determine the number of capacity misses of a program. A capacity miss does not necessarily cause misses in every memory layer, e.g., if a block has been removed from the level 1 cache, then it may still reside in the level 2 cache.

Conflict misses occur when a referenced block can only be placed in an occupied cache line. This type of misses can occur in caches that are not fully associative, even though there are still empty cache lines. This is due to the fact that in set-associative caches more than one block map to the same cache line. In a fully-associative cache conflict misses do not occur, since any block can reside in any cache line. If a fully-associative cache has no empty cache line in which a block can be placed, then a capacity miss occurs and not a conflict miss.

2.1.4 Virtual Memory

The virtual-memory system can be viewed as a caching system running in parallel to the memory hierarchy described so far [35] (see Figure 2.3). This parallel system handles data transfers between the primary and secondary memory. Its purpose is to make multiple programs that run simultaneously on the computer think that they each have the entire main memory

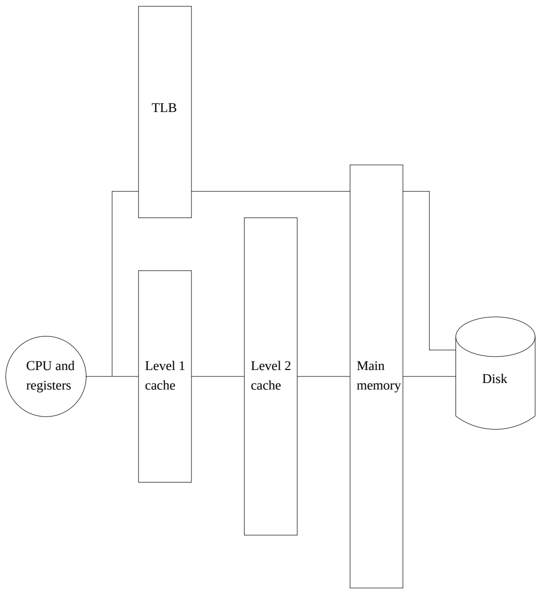


Figure 2.3: Virtual memory makes up a memory hierarchy in parallel to the hierarchy of Figure 2.1.

at their disposal. A program is unaware that it actually shares the main memory with other programs, and as a consequence, the virtual memory system removes the burden from the programmer of managing main memory manually. This means that the virtual-memory system has to ensure that sharing of main memory is done in a safe way, so that one program cannot overwrite the data of another program.

Knowing the exact details of the virtual memory system is not that important in the scope of this thesis. Nevertheless, a few concepts are worth noting:

Virtual and *physical addresses.* A distinction is made between virtual addresses referring to imaginary storage and physical addresses referring to memory represented by DRAM chips. Since each process should be able to execute as if it had the entire main memory at its disposal, each process has its own virtual address space. The size of a pointer is 32 bits on 32-bit computers, so the number of memory addresses that can be referenced is limited to 2^{32}, i.e., the virtual address space is 4GB. As many processes often run simultaneously, and their total address space is most likely much bigger than the amount of physical memory available, at any time the virtual-memory system manages which of the virtual pages that are currently present in physical memory and which reside on disk. As such, the amount of physical memory available does not directly restrict how much memory a single process can allocate. On a computer with a small amount of physical memory the virtual-memory system is simply forced to keep a larger portion of the allocated memory on disk than would be the case on a computer with a large amount of physical memory. A consequence of having only a small amount of physical memory available is therefore more page faults.

Address translation and *page tables*. When a process references some data, the virtual address must be translated into a physical address. Since main memory may be shared between multiple programs, it most often contains only a subset of the whole virtual address space addressable by a given process, so data pointed to by virtual addresses may actually reside on disk. Each process therefore has a page table containing a mapping from virtual addresses to physical addresses. The page table contains information about which virtual pages that are currently in main memory and which that are on disk. The page tables themselves are kept in main memory, but as any other part of main memory a page table can be temporarily written to disk, if the operating system has intentions of using the space it occupies for something different.

Translation look-aside buffer (TLB). Like a page table the TLB contains a mapping from virtual to physical addresses. It acts as a cache for page tables by containing the most recent address translations. The TLB is implemented in hardware to ensure that virtual pages often referenced (due to temporal locality of reference) are translated quickly.

When the CPU references some data, the virtual address of the data is looked up in the TLB and in the level 1 cache simultaneously. While the level 1 cache checks if it contains the cache block with the requested virtual address, the TLB translates the virtual address into a physical address and hands the physical address over to the level 1 cache. If there is a hit on the virtual address in the level 1 cache, then the cache uses the physical address that it got from the TLB to check if the hit cache block is actually the correct one, or if it belong to another program and just by coincidence shared the same virtual address.

2.2 Memory Latency

Until now, we have used the term *memory latency* to describe the overhead incurred by the memory system on the running time of programs. But what is memory latency, from where does it originate, and how does it affect memory design?

Latency is the period of time that one component spends waiting for another component, in other words, latency is wasted time. So memory latency refers to the CPU time wasted waiting for the memory system. The impact of memory latency is often measured in the number of clock cycles that the CPU has to wait. This is practical for two reasons:

- Clock frequencies of memory and CPU seldomly match. Most often the memory clock frequency is lower than the CPU clock frequency, and the same CPU model is often shipped with differing clock frequencies but with the same memory clock frequency. So even though measuring latency in memory clock cycles provides a measure that is independent of the CPU, it is not very useful. Memory latency is interesting in relation to the CPU performance.

- Depending on the degree of instruction-level parallelism modern CPUs are able to execute a number of instructions every clock cycle. If we know this degree and the memory latency in CPU clock cycles, then it is easy to interpret the importance of a given latency by simply calculating the number of instructions that could have been executed during the waiting period.

Memory latency is a non-uniform metric, i.e., it is specific to a particular layer of the hierarchy and increases when moving from left to right in Figure 2.1. Therefore, each layer in the memory hierarchy adds to the total memory latency. Assuming that data are not read in parallel, for each layer, the cost is the sum of the time spent finding the requested block and the time spent transferring it to a faster layer. A data request that causes cache misses all the way through the hierarchy therefore incurs latency corresponding to the sum of the latencies of all layers.

The latencies of the memory systems of the Intel® Pentium® and Sun UltraSPARC™ computer families were investigated by Olsen & Skov [35]. To give an idea of the actual impact of memory latency and how latencies differ among memory layers we summarize the results of their investigation in Figure 2.4. As expected, the figure shows that latency increases the further away we get from the CPU.

Memory layer	Latency	
	Min	Max
Level 1 cache	2	3
Level 2 cache	7	23
Main memory	41	262

Figure 2.4: Minimum and maximum memory latencies in CPU clock cycles on a variety of different computers measured by Olsen & Skov [35].

The distinct difference in minimum and maximum latencies of main memory actually provide empirical evidence for the CPU-DRAM performance gap: At the main memory layer the latency of the Pentium® family was measured to 41, 73, 86, and 262 for the Pentium® , Pentium® II, Pentium® III, and Pentium® 4 respectively — almost a fourfold increase over a ten year period. For the UltraSPARC™ family the latencies have tripled from the UltraSPARC™ I to the UltraSPARC™ III.

Furthermore, the investigation shows that while the latency of level 1 caches has been rather constant during the past ten years, the level 2 caches of the Pentium family have become faster, whereas those of the UltraSPARC™ family have become slower. The success of Intel's level 2 caches contrasts the general tendency of increasing memory latency. The success is due to the fact that the level 2 caches are built using faster technologies than DRAM and that the level 2 cache has recently been integrated into the same chip as the CPU to ensure faster communication.

Olsen and Skov used a tool called *LMbench* [10] to measure memory latencies. We have used the same tool to obtain the latencies of the benchmarking computers at our disposal. The characteristics of these computers are shown in Figure 2.5. For the AMD computer LMbench reported the latency of the level 1 cache to 2 ns, the latency of the level 2 cache to 12 ns, and the latency of main memory to 157 ns. LMbench reported 1 ns, 10 ns, and 170 ns respectively for the Pentium® computer.

CPU	Memory system
AMD Athlon XP 2100+ (1.73 GHz) Model 6	64 KB 2-way L1 cache, 64-byte cache lines. 256 KB 16-way L2 cache, 64-byte cache lines. 256 MB main memory.
Pentium® 4 Northwood 1.8 GHz	8 KB 4-way L1 cache, 64-byte cache lines. 512 KB 8-way L2 cache, 64-byte cache lines. 512 MB main memory.

Figure 2.5: The characteristics of our benchmarking computers.

The time that the CPU spends waiting for the memory system is not necessarily completely wasted. If instruction i_{miss} causes a cache miss and the execution of the following instruction, i, does not depend on the result of instruction i_{miss}, then the CPU can execute instruction i while waiting for the memory system. Of course, if multiple independent instructions follow i_{miss}, then the amount of wasted time can be further decreased.

The latency caused by a certain memory layer is influenced by that layer's degree of associativity. At the memory layers closest to the CPU the associativity is implemented in hardware, and higher associativity means more complex circuits and therefore higher latency. The fewer cache misses induced by higher associativity simply does not pay for the increased memory latency. Along with the results of [37], stating that for sufficiently large caches the degree of associativity has minor impact on the number of cache misses (page 11), the higher latency makes it infeasible to use full associativity at the closest memory layers.

Other parts of the computer than the memory system may cause latency. Pipeline hazards in the CPU due to branch mispredictions or other dependencies among instructions may force the pipeline to flush completely or stall for a few cycles. Also, communicating with external devices such as graphic and network devices causes latency.

2.3 Cache Misses and Latency by Example

By now it should be clear that being aware of memory latency and cache usage when designing algorithms may affect running time in a positive way. But just how severe are the effects of careless algorithm design and how do the different kinds of cache misses occur in practice? By use of a simple example we will answer these questions.

In [11] Bojesen et al. implemented a simple but clever C-program to measure the performance of the level 1 cache on a number of different computers. As such, the program was designed to reveal the same characteristics of the memory system as those investigated by Olsen & Skov. Nevertheless, we can use this C-program in a different way than originally intended by Bojesen et al. to investigate how an algorithm that makes good usage of the memory hierarchy can outperform an algorithm that uses the memory hierarchy inefficiently.

The program computed the sum of N integers placed contiguously in an array. Depending on the value of a variable *step* the integers were referenced according to different patterns. Program 1 is the summing program used by Bojesen et al.

```
unsigned int sum(unsigned int* a,
                 unsigned int step,
                 unsigned int N) {
  unsigned int i;
  unsigned int mask = N - 1;
  unsigned int result = a[0];

  for (i = step; i != 0; i = (i + step) & mask)
    result += a[i];

  return result;
}
```

Program 1: The summing program of Bojesen et al.

With a step value of 1 the integers were referenced sequentially and with a value of p, chosen to be the smallest prime larger than the block size, the reference pattern ensured that no two subsequent references caused access to the same cache line. By calculating the index of the ith integer to visit as $i * step \bmod N$ and choosing N to be a power of 2 the values of 1 and p of *step* ensured that all integers were referenced exactly once[2].

With any of the two values of *step* the program executed the exact same number of instructions, so the difference in running times was solely due to the different reference patterns.

With B denoting the number of integers fitting in a block, the program caused N/B level 1 cache misses with a *step* value of 1 and N level 1 cache misses with *step* set to p. By the benchmarks of Bojesen et al. the summing program turned out to be in the range of 3.3 to 10.1 times faster when causing only N/B cache misses than when causing N. The variation among the relative efficiencies were due to varying block sizes and differences in size and associativity of the caches on the computers considered.

We have tested the summing program on our benchmarking computers (Figure 2.5). For an input array exceeding the size of the level 2 cache the program executed 5.8 times faster on the Athlon computer when causing N/B cache misses than when causing N cache misses. On the Pentium® 4 the fewer cache misses made the program 3.4 times faster[3].

In Figure 2.6 the reference patterns associated with the two values of *step* are illustrated on an idealized cache. The cache on the figure is fully associative and each of its eight cache lines contains a single block of eight integers.

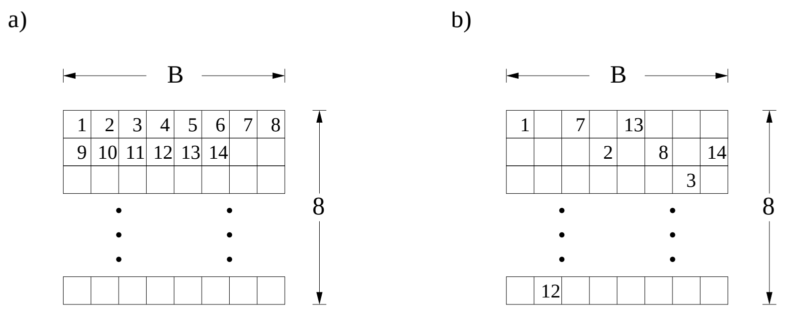


Figure 2.6: The numbering marks the order in which the caches are accessed when N is equal to or larger than the capacity of the cache. a) The efficient reference pattern accesses the cache continuously, i.e., *step* is 1. b) The inefficient reference pattern accesses the cache in steps of p, where p is the smallest prime larger than the block size B.

It may not be obvious how the bounds of N and N/B cache misses

[2]The reader can verify this by noting that the prime factorization of any integer is unique and that for powers of 2 the only prime in the factorization therefore obviously is the prime 2 itself.

[3]The execution times were measured in CPU time.

are obtained. So, in the analysis that follows we will prove these bounds informally. As in Figure 2.6 we assume a fully associative cache and we let M denote its capacity. The cache uses the LRU replacement policy:

$\mathit{step} = 1:$ Initially the cache is empty. So as long as data fit in the cache, i.e., N is smaller than or equal to M, the continuous reference pattern incurs a compulsory miss for every B references. Hence, the summing program causes N/B cache misses when N is smaller than or equal to M.

If N is larger than M, then an additional $N-M$ integers are referenced. Since the cache contains no empty cache lines when these integers are referenced, an additional capacity miss is incurred for every B references for a total of $(N - M)/B$ capacity misses.

If we do not consider the nature of the cache misses, i.e., if they are compulsory misses or capacity misses, then the summing program causes N/B cache misses regardless of the input size.

$\mathit{step} = p:$ The choice of p as the smallest prime larger than the block size guarantees that no two consecutive integer references access the same cache line. By choosing M and B as powers of 2, as Bojesen et al. did, the program still causes only N/B cache misses as long as data fit in the cache. But instead of incurring a miss for every B references (as for $\mathit{step} = 1$) the program now causes compulsory misses when referencing the first N/B integers. The remaining $N - N/B$ integers can be referenced without any further misses.

If data exceed the capacity of the cache, then the situation becomes more complicated. First we notice that after the first M/p references the cache is full and we are about to access a block that has not been in cache earlier (remember that N now is larger than M). A capacity miss occurs and according to the replacement policy this miss causes the block that was accessed as the very first to be evicted from cache, i.e., the block least recently used.

Depending on the size of N we now have one of the two following situations for the next integer reference: Either the next integer relies in a block that has not been in the cache earlier (i.e., we have not yet reached the end of the integer array), or we have reached an index exceeding the size of N (i.e., we are back at the beginning of the array). Both situations cause capacity misses. The former situation evicts a block just as when the preceding integer was referenced. The latter also causes a capacity miss, since the block to reference (which is the one containing the beginning of the integer array) has just been replaced. In fact, even the very preceding reference may have caused its eviction.

Hence, when N is bigger than M, every reference causes a cache miss and we have the bound of N cache misses.

Depending on the value of *step* the program was constructed with the intent of making as many or as few cache misses as possible, so the running times represent extreme usage of the cache. A programmer unaware of the overhead incurred by memory latency seldomly would make programs exhibiting a reference pattern as inefficient as that associated with the slow summing program. Nevertheless, the investigation clearly shows that taking memory latency into account when designing algorithms may decrease running time with a constant factor that is significant. Analysis of the program in the RAM-model would not have revealed this behavior.

2.4 Summary

In this chapter we have presented the key concepts of modern memory systems and observed how the hierarchy adheres to the principle of locality of reference. We have described how the different layers of the memory hierarchy are organized and seen that parameters such as block size, associativity, and replacement policy influence how data are moved between various memory layers and how different types of cache-misses occur. Also, we have noticed that the virtual memory system can be seen as a hierarchy running parallel to the traditional memory hierarchy.

We now know what memory latency is and how it can influence the running time of programs. With the summing program we saw the effects of not paying attention to the principle of locality of reference.

CHAPTER 3

Memory Models

"Do not quench your inspiration and your imagination; do not become the slave of your model."

— Vincent Van Gogh

In Chapter 2 we saw how the memory access pattern of an algorithm highly influences its running time in practice. As traditional work complexity analysis in the RAM-model cannot capture these access patterns, we need more elaborate computational models that can.

In this chapter we review three computational models of modern memory systems, namely the external-memory model, the hierarchical memory model, and, most importantly, the ideal-cache model.

The external-memory model is described in Section 3.1 and is the model of choice in the design and analysis of external-memory algorithms. The model is interesting since it in some ways resembles the ideal-cache model. Nevertheless, the external-memory model has some drawbacks that make it insufficient for analyzing algorithms in a cache-oblivious way.

Though the hierarchical memory model is not that widely used we have included its description in Section 3.2 to provide an example of how multilayered memory systems were modeled prior to the introduction of the ideal-cache model.

In Section 3.3 we describe the ideal-cache model. Though the model at first glance may seem very simple compared to real memory systems, it is both theoretically and practically justifiable. We investigate these justifications in Section 3.4.

3.1 The External-Memory Model

The idea demonstrated in the summing example (Section 2.3) of analyzing algorithms by counting the number of memory accesses it causes is not new. In the area of external-memory algorithms and data structures, complexity analysis involving the count of memory accesses has been known for many years.

External-memory algorithms and data structures deal with data sets the size of which exceeds that of main memory. The idea is that by managing data placement and movement between internal and external memory explicitly, the impact of latency associated with disk accesses can be minimized. As such, these algorithms and data structures bypass the virtual memory system.

A consequence of the explicit data management is that the caching characteristics of internal memory can be optimized for a specific algorithm, e.g., the replacement policy may be specialized — or even changed during the execution of a program. It is all up to the programmer.

According to a recent survey by Vitter [41] on external-memory algorithms and data structures the first work on complexity analysis in the area dates back to the PhD thesis on sorting by Demuth in 1956. To facilitate the analysis of external-memory programs a collection of reasonably accurate models of the memory system's characteristics has evolved throughout the past 40 years. The most widely used is the *external-memory model* formalized by Aggarwal and Vitter in 1988 [3]. The model operates on two consecutive layers of memory, namely an internal and an external one. The *I/O-complexity* of an algorithm is measured in the number of memory accesses or I/Os it performs on the external memory layer. Since the external memory is often a disk, the external-memory model is also known as the *I/O model* and the *disk access model*.

The external-memory model uses the following parameters, some of which were already used in the analysis of the summing program:

N: the number of elements that is to be processed by the algorithm,

M: the number of elements that fits in internal memory,

B: the number of elements that can be transferred in a single block,

P: the number of blocks that can be transferred concurrently.

The parameter P models a special feature that a disk might have, such as multiple I/O channels and read/write heads. P can also model multiple disks that can transfer data concurrently. Figure 3.1 depicts the external-memory model.

It is assumed that $1 \leq B \leq M < N$ and $1 \leq P \leq \lfloor M/B \rfloor$. The former inequality states that a block contains at least 1 element and that

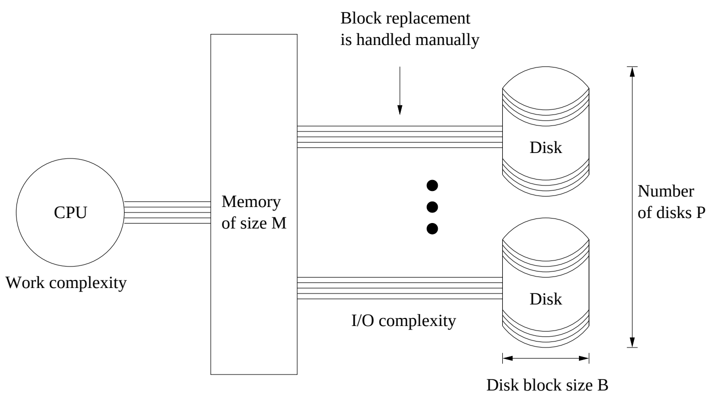


Figure 3.1: The external-memory model is a two-layered memory model consisting of an internal and an external memory. Block replacement is managed explicitly, i.e., the algorithm itself dictates the replacement policy, so the knowledge of B and M is crucial.

the size of the problem has to exceed the size of internal memory. The latter inequality states that the number of blocks that can be transferred concurrently should not exceed the number of blocks in memory. This is a rather obvious restriction since it makes no sense to transfer more blocks concurrently than that there is room for in internal memory.

Often the work complexity of an external-memory algorithm is of interest as well as the I/O complexity, because explicit memory management usually causes an external-memory algorithm to be more complex than its more conventional competitors. As for ordinary algorithms the well-known RAM model is most often used for the work complexity analysis.

A drawback of the external-memory model is that even though knowledge of the exact values of B and M is not strictly necessary when designing algorithms in the model, these values must be known when the algorithms are implemented. Remember, the whole idea of external-memory algorithms is to tune the algorithms to the characteristics of the memory system by moving data manually. As the parameters B and M may vary from one memory system to another, they restrict an implementation to work efficiently only on a specific memory system. Porting the programs to other platforms becomes a non-trivial task.

In the memory layers closest to the CPU, i.e., the level 1 and level 2 caches, the replacement policies are implemented by hardware, so data placement and movement cannot be managed explicitly. However, this does not mean that the external-memory model is not applicable to these layers. Knowing B and M and being aware of the actual replacement policy of those layers still makes it possible to adjust an algorithm. As can be imagined, optimizing an algorithm to multiple layers of a memory system with the

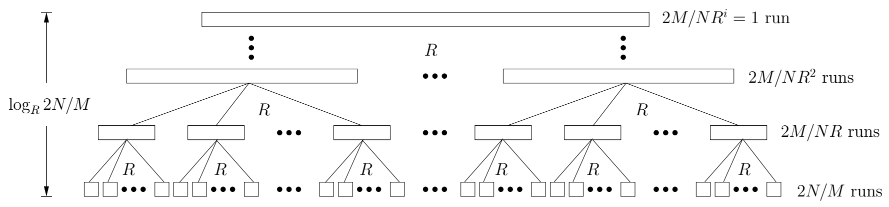


Figure 3.2: In the merging phase of multiway mergesort the initial $2N/M$ runs are reduced to a single run in $\log_R 2N/M$ consecutive merges.

values of B and M varying from one layer to another can be a tedious job.

Since the knowledge of B and M is crucial in the external-memory model, it is also known as the *cache-aware model* or *cache-conscious model.*

3.1.1 External-Memory Algorithms by Example

To see how implementations of external-memory algorithms depend on the parameters of the memory system we consider a multiway mergesort algorithm for external memory based on a presentation by Jeff Vitter [42] at the EFF Summer School on Massive Data Sets[1]. The task is to sort N elements residing on disk where N is much larger than M.

Multiway mergesort is a two-phase algorithm:

Run-formation phase: In the run-formation phase the $\frac{N}{B}$ blocks are read into internal memory in chunks of $\frac{M}{2B}$ blocks, i.e., one half memory load at a time. The chunks are sorted following some traditional sorting algorithm that works well in internal memory, whereupon they are written back to disk. When the run-formation phase is completed, we have $\frac{2N}{M}$ sorted *runs* residing on disk. Each run contains $\frac{M}{2B}$ blocks.

We read in the N elements in half memory loads rather than full memory loads to be able to overlap computation and I/O, that is, while one half memory load is being sorted the previously sorted half memory load is outputted.

Merging phase: In the merging phase we repeatedly merge together $R = \frac{M}{2B} - 2$ runs at a time in an on-line manner. That is, we perform a number of R-way merges until all of the $\frac{2N}{M}$ runs are merged. Each of the R-way merges are performed in an on-line manner by only having one block of each run in memory at a time. As such, internal memory acts as a buffer for the runs residing on disk. Each R-way merge results in a new run of size $\frac{RM}{2B}$. Therefore, when all N elements have been merged once, the $\frac{2N}{M}$ runs of size $\frac{M}{2B}$ are reduced to $\frac{2N}{MR}$ runs of size

[1]Held June 17–July 1, 2002 at BRICS, University of Aarhus, Denmark.

$\frac{RM}{2B}$. Each pass over the N elements in the merging phase results in increasingly longer runs, so after $\log_R \frac{2N}{M}$ passes only 1 run is left and we are done.

By choosing $R = \frac{M}{2B} - 2$ we can overlap the merging process with disk I/O as was the case in the run-formation phase. We let the R runs stream through internal memory by use of R double buffers, each of size $2B$ (i.e., one double buffer consists of 2 buffers of size B each). The one half of the double buffer is filled up with elements from disk while the elements of the other half of the double buffer are being merged.

The double buffers do not take up the entire internal memory. The -2 term of R leaves space in memory for an output buffer to which the elements are merged and held temporarily before they are written blockwise back to disk.

The merging phase is depicted in Figure 3.2. The following theorem states the I/O-complexity of the algorithm.

Theorem 1. *The I/O-complexity of the external memory multiway merge-sort algorithm is* $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

Proof. The run-formation phase passes over the elements only once. The merging phase passes over the elements once for each of the $\log_R \frac{2N}{M}$ levels in the merge-tree. Hence, the total number of passes over the elements is

$$\begin{aligned} 1 + \log_R \frac{2N}{M} &\approx 1 + \log_{M/2B} 2\frac{N/B}{M/B} \\ &\approx 1 + \log_{M/B} \frac{N}{B} - 1 \\ &\approx \log_{M/B} \frac{N}{B}. \end{aligned}$$

Each pass read the N elements into memory and writes them back to disk once, so the I/O-complexity is proportional to

$$2\frac{N}{B}\log_{M/B}\frac{N}{B} = O(\frac{N}{B}\log_{M/B}\frac{N}{B}).$$

□

The I/O-complexity of the multiway mergesort algorithm is in fact equal to the optimal sorting bound for comparison-based sorting in the external-memory model [3].

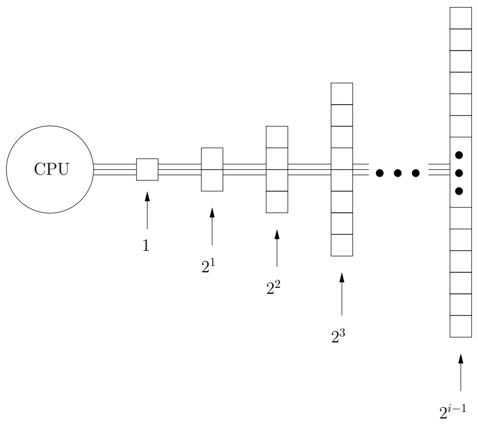


Figure 3.3: For cost function $f(i) = \lceil \log_2 i \rceil$ the hierarchical memory model consists of several memory layers of polynomially increasing size.

3.2 The Hierarchical Memory Model

In contrast to the two-layered external-memory model the hierarchical memory model of Aggarwal et al. [1] considers memory hierarchies of multiple layers.

The hierarchical memory model resembles the RAM model. It accepts the same operations and also assumes a potentially unlimited number of memory registers $R_1, R_2, \ldots$ each having space for one integer. As with the RAM model it is the running time of an algorithm that is of interest, but in contrast to the RAM model accessing location R_i takes $f(i)$ time instead of constant time. The function f is assumed to be monotonically increasing and the choice of f determines the size and numbers of memory layers. Typically the functions $f(i) = x^\alpha$, where $\alpha > 0$, and $f(i) = \lceil \log_2 i \rceil$ are chosen [1] — the latter indicating a memory hierarchy of several layers whose sizes increase by a factor 2 at each layer. Notice, though, that these choices of f are just examples and other functions that one finds appropriate can be applied. On Figure 3.3 the hierarchical memory model is depicted for $f(i) = \lceil \log_2 i \rceil$.

As such, the hierarchical memory model mimics the behavior of a memory hierarchy consisting of increasingly larger amounts of increasingly slower memory layers, but the model has some shortcomings. First of all, it fails in modeling varying degrees of associativity among memory layers. Secondly, it assumes that data are explicitly moved between layers by the programmer, but in practice the programmer has no such control over data management in the faster layers. Thirdly, and most importantly, choosing a function f that makes the model realistic is a difficult task, as it requires detailed knowledge of the characteristics of the memory system. Therefore, it is hard to make running times predicted by the model match those observed in practice, and if one should succeed in this task, f will most likely be highly dependent on the specific memory system for which the algorithm was designed [41].

Aggarwal et al. later extended the model to handle block transfers [2]. Though this made the model more realistic, Aggarwal et al. still viewed the model only as a beginning in the theoretical exploration of memory hierarchies.

3.3 The Ideal-Cache Model

Analyzing algorithms in the ideal-cache model is very similar to analyzing algorithms in the external-memory model. It involves the same parameters B, M and N (see page 22) but the ideal-cache model does not handle concurrent transferring of multiple blocks. Therefore, the parameter P is omitted. As was the case with external-memory algorithms we are interested in analyzing how many memory transfers an algorithm incurs. In a cache-oblivious context we use the terms I/O-complexity and cache complexity interchangeably when we analyze memory behavior. For cache-oblivious algorithms, ordinary analysis of work complexity is done in the RAM model. We denote the number of cache misses incurred by a cache-oblivious algorithm by $Q(N, M, B)$[2], if the algorithm runs on a cache of size M, block size B, and takes input of size N. We use $W(N)$ to denote the work complexity of a cache-oblivious algorithm. In the scope of cache-obliviousness the term cache does not only refer to level 1 and 2 caches but to the smaller of any two consecutive memory layers.

According to Prokop [38] 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". By this description all algorithms that do not pay special attention to the memory system are cache-oblivious. As a consequence, all traditional algorithms designed in the RAM model are cache-oblivious. Therefore it makes sense to distinguish between cache-oblivious algorithms and *optimal* cache-oblivious algorithms, i.e., cache-oblivious algorithms that incur an asymptotically minimum number of cache misses. Since algorithms designed in the ideal-cache model are cache-oblivious the model is also known as the *cache-oblivious model*.

3.3.1 Consequences of Obliviousness

At first sight, being unaware of B and M might seem like a restriction. But it has some surprisingly powerful consequences that actually deal with the two

[2]We may just use $Q(N)$ to ease notation.

main objections to the external-memory model, i.e., platform dependency and complex programming for multiple caching layers.

The first consequence of obliviousness is that, if a cache-oblivious algorithm uses two consecutive memory layers optimally, then it must automatically use *any* two consecutive layers optimally. This means that cache-oblivious algorithms that behave optimally in a two-layer memory hierarchy will also behave optimally in all layers of a memory hierarchy of more than two layers.

The second consequence is that cache-oblivious algorithms automatically tune to the system in which they run. A cache-oblivious algorithm that works well on one computer should therefore work well on any computer — a feature that completely eliminates portability issues. This self-tuning ability is a clear advantage over memory efficient algorithms that require knowledge of the memory system that is not always available from the manufacturer and may be difficult to extract automatically.

However, not knowing B and M poses a problem as data placement and movement between memory layers can no longer be managed by the programmer. The programmer no longer has control over the replacement policy, so how can she possibly know when a cache-oblivious algorithm uses the memory hierarchy optimally? This question and the two mentioned consequences of obliviousness are dealt with in the computational model of cache-oblivious algorithms, namely the ideal-cache model that, as its name suggests, assumes an *ideal cache*.

3.3.2 The Ideal Cache

The ideal cache is fully associative and relies on the optimal replacement policy, i.e., the block that is accessed furthest in the future is replaced in case of a cache miss. The movement of data between the layers is done automatically. The model has only two memory layers, a main memory layer that is assumed to be arbitrarily large and a data cache containing M words. The cache is divided into M/B cache lines each containing B words[3]. Furthermore, the cache is assumed to be tall, that is, the size of a cache line is no wider than the number of lines the cache can contain in total. The *tall-cache assumption* can be stated as $M = \Omega(B^2)$. The ideal-cache model is depicted in Figure 3.4.

The ideal-cache model offers an obvious advantage of simplicity over the external-memory model, but some assumptions are made in the model that may not seem reasonable compared to the characteristics of real world caches and memory systems. Firstly, the optimal replacement policy is unrealistic as it requires the knowledge of future program execution. Secondly, very few modern memory systems have only two layers of memory. Finally,

[3]Frigo et al. [19] denote the cache size by Z and the cache line length by L. We prefer using M and B to emphasize the connection to the external-memory model.

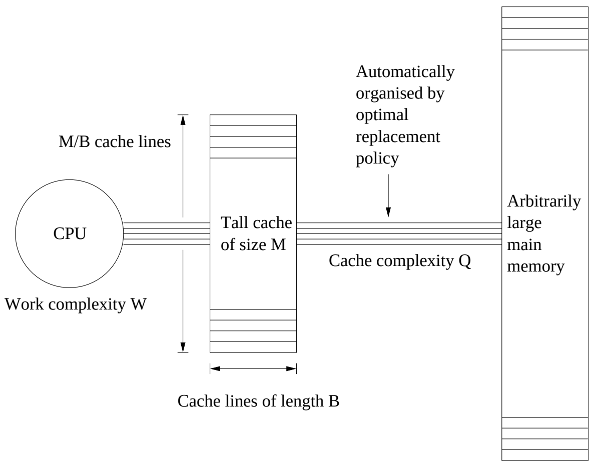


Figure 3.4: The ideal-cache model is a two-layered memory model. The cache is fully associative and data is transported between cache and main memory automatically following an optimal replacement policy. B and M are unknown but the cache is assumed to be tall, i.e., $M = \Omega(B^2)$.

full associativity is seldomly supported. Most caches have a limited degree of associativity. In Section 3.4 we will verify that these assumptions are reasonable.

3.3.3 Cache-Oblivious Techniques

According to Demaine [18], the two main techniques for designing cache-oblivious algorithms are sequential scanning, that inherently exhibits good spatial locality, and divide-and-conquer. Intuitively, these two techniques are well-suited because they adhere to the principle of locality of reference.

In the external-memory model scanning an array of N elements causes $\lceil N/B \rceil$ blocks to be read from disk into internal memory. In the ideal-cache model the same task causes at worst $\lceil N/B \rceil + 1$ cache misses. This difference is due to not knowing B and M for cache-oblivious algorithms. A consequence of this is that we cannot align the array with the boundaries of the cache lines. Figure 3.5 illustrates this issue of alignment for the scanning example. It follows from the example that accessing a continuous segment of memory that has the size of a single block in the worst case means accessing two physical blocks.

Most of the algorithms originally presented by Frigo et al. [19] follow the divide-and-conquer approach. Traditionally, by this approach an algorithm repeatedly refines the problem size until some base case is reached that can be solved easily. For example, in traditional 2-way mergesort an array of elements is repeatedly divided into two subarrays each of half the size. The

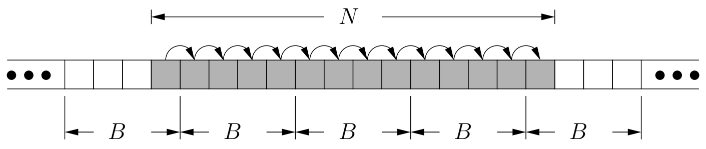


Figure 3.5: Due to the possibly bad alignment cache-oblivious scanning of an array of N elements incurs $\lceil N/B \rceil + 1$ cache misses in the worst case.

base case is reached when the subarrays only contain one element each, and therefore can be merged easily.

In a cache-oblivious context, divide-and-conquer also means splitting up the problem into smaller parts. In analogy to the mergesort example, the base case is reached when the subproblem becomes easily solvable. But where easily solvable for traditional algorithms (i.e., in the RAM model) means solvable in a constant number of *instructions*, by the cache-oblivious approach it means that solving a subproblem will cause no further *cache misses*. Typically, this is when the subproblem fits in cache[4].

Cache-oblivious algorithms following the divide-and-conquer approach will often exhibit cache complexities that are optimal within a constant factor — especially when the divide-and-conquer cost is dominated by the leaf cost, i.e., when the number of leaves in the recursion tree is polynomially larger than the divide and combine cost in terms of cache misses [18].

3.4 Justifying the Ideal-Cache Model

The ideal cache is an abstraction of real memory systems. Compared to the characteristics of modern memory systems, like those presented in Chapter 2, the characteristics of the ideal cache clearly make the ideal-cache model a simplification of a real memory hierarchy.

The number of cache misses an algorithm causes in the ideal-cache model is within a constant factor of the number of cache misses the algorithm causes in a real memory system [19]. This is a key property of the model. In this section we will investigate the validity of the ideal cache and, more importantly, look into what causes the constant factors that separate the cache complexities analyzed in the ideal cache from the cache complexities that can be observed in real memory hierarchies.

If we want to predict the cache complexity of algorithms in real memory systems accurately, then it is a must to have good knowledge of how the

[4]Note, that this does not mean that the algorithm stops dividing the problem when it fits in cache (this would make the algorithm cache-aware), but merely that this point in the divide-and-conquer recursion provides for the base case of our cache complexity analysis.

ideal cache differs from these systems, and how the differences affect cache complexity.

Theoretically, the assumptions of the ideal-cache model are justified by Frigo et al. They show that by a number of reductions the ideal cache can be modified into a more realistic cache model assuming LRU replacement, multiple layers of memory, and direct mapping. We will not formally prove the reductions here but describe the reductive steps in an informal way to explain how weak or strong the ideal-cache model is compared to real memory systems.

3.4.1 Assumption: Optimal Replacement

Frigo et al. use an older result of Sleator and Tarjan [39] on the efficiency of a variety of on-line replacement strategies, such as LRU and FIFO, relative to an optimal off-line strategy[5] to justify the reduction of optimal replacement to LRU or FIFO replacement.

Sleator and Tarjan show that for any constant factor $c > 1$, on an LRU or FIFO-cache of size M, any algorithm incurs at most c times as many cache misses as the same algorithm would incur on an optimal cache of size $(1 - 1/c)M$. We can express this as

$$Q_{LRU}(N, M, B) = cQ_{OPT}(N, (1 - 1/c)M, B),$$

where Q_{LRU} is the number of cache misses with the LRU replacement and Q_{OPT} is the number of cache misses with the optimal replacement. Frigo et al. choose $c = 2$ and can therefore rightfully argue that an algorithm that causes $2Q$ cache misses on an LRU-cache of size M and block size B causes at most Q cache misses on an optimal cache of half the size $(M/2)$ and the same block size. Hence, LRU replacement is just as good as optimal replacement up to a constant factor of cache misses and a constant factor of wasted cache lines.

Frigo et al. then define the cache complexity of an algorithm as being *regular* if it satisfies the condition $Q(N, M, B) = O(Q(N, 2M, B))$. This regularity condition states that when the number of cache lines is halved, then the cache complexity is affected only by a constant factor. The condition is important since it can be used to guarantee that an algorithm does not exhibit worst-case cache behavior when using the LRU replacement.

So, what is meant by worst-case LRU behavior, and which algorithms are of irregular cache complexity? Consider an algorithm that cyclically scans an array of $M/B+1$ elements — assuming that the $M/B+1$ elements reside in $M/B+1$ distinct blocks. After scanning the first M/B elements the cache will be full, so accessing the last element of the array causes a cache block

[5] An on-line strategy has no knowledge of future memory references, whereas an off-line strategy know the entire sequence of references in advance.

to be evicted. The LRU policy will evict the block holding the first element of the array, since that block is the one least recently used. However, the next element to be accessed resides in the block just evicted, so a cache miss occurs and another block must be evicted. The LRU policy evicts the block holding the second element of the array, which is the next to be accessed. This pattern where every access will cause a cache miss continues until the algorithm somehow terminates. In other words the cache complexity depends on how many passes over the elements the algorithm incurs.

Doubling the number of cache lines to $2M/B$ would only cause cache misses for the first $M/B+1$ accesses, whereas all subsequent accesses would be to blocks already in the cache — no matter how many times the algorithm cycles through the array. Now the cache complexity depends on M and B rather than the number of passes over the elements. Therefore, this worst-case LRU behavior is not regular, so this behavior is impossible for algorithms of regular cache complexity bounds.

Combining the results of Sleator and Tarjan with the regularity condition yields Corollary 13 of [19] stating that any algorithm with a regular cache complexity on an ideal cache of size M will have an asymptotically equal cache complexity on an LRU cache of the same size.

The analysis of Frigo et al. assumes real LRU replacement. In real memory systems LRU is often approximated or a random replacement strategy is used instead, so their Corollary 13 is not entirely true to real memory systems. But, as we saw on page 11, the choice between LRU and random replacement only has a minor influence on cache performance in practice — at least for caches of a certain minimum size. However, one should be aware that the practical observations only indicate similar average case behavior of the two replacement strategies. The way in which the LRU strategy supports the principle of locality of reference may very well make it superior to the random strategy when it comes to cache-oblivious algorithms.

3.4.2 Assumption: Two Memory Layers

In real memory hierarchies there are 3 to 5 layers (not counting the CPU registers), so we need to be sure that the two-layered model is sufficient. In other words: A cache-oblivious algorithm of optimal cache complexity in the two-layered ideal cache is also of optimal cache complexity in an LRU memory hierarchy of multiple layers. We can argue this in two ways:

- By assuming that all layers in the memory hierarchy follow the inclusion property and are managed by the optimal replacement policy, this is intuitively true. Recalling Section 2.1, the inclusion property states that data cannot be present at layer i unless also present at layer $i+1$ (layer i is closer to the CPU than layer $i+1$). By definition, the optimal replacement policy will ensure a minimum number of cache misses in each layer. As a consequence the whole hierarchy will be used optimally. If we now apply Corollary 13 of Frigo et al. at each layer of the hierarchy, then the optimal replacement policy is turned into the LRU policy. This reduction increases the cache complexity at each layer by a constant factor depending on the algorithm under consideration.

- Frigo et al. show the asymptotic optimality of the multilayered LRU hierarchy in a more formal way. First they argue that when using LRU replacement in hierarchies of multiple layers the inclusion property is maintained during both cache hits and cache misses. This argument is important, since if the inclusion property was *not* maintained within multiple LRU layers, then the only reliable preservation of the principle of locality would be that of the memory layer closest to the CPU.

  Secondly they argue that, at any layer i in a hierarchy of multiple layers and LRU replacement, cache hits happening at lower layers are not seen. In fact, layer i acts exactly as if it was the first layer in a two-layer hierarchy. That is, for any sequence of accesses it contains exactly the same data as it would have contained if it was the first layer in a two-layer model that had served the same sequence of accesses. As a consequence, the cache behavior at each boundary between any two consecutive layers in the memory hierarchy can be analyzed in the ideal-cache model.

  Therefore, the ideal-cache model applies at each boundary between two layers in a multilayered LRU hierarchy. And, by Corollary 13 of Frigo et al., each layer can be turned into using the LRU policy with an increase in cache complexity of a constant factor for each layer.

As was the case with the reduction from optimal replacement to LRU replacement, removing the assumption of only two memory layers increases the cache complexity observed at each caching layer by a constant factor. This constant factor depends on the regularity of the algorithm under consideration.

3.4.3 Assumption: Auto Replacement and Full Associativity

The ideal cache assumes that cache blocks are automatically replaced in case of cache misses. From a programmer's point of view this actually is a viable assumption: Between the smaller memory layers block replacement is handled automatically by hardware, whereas data transfers between main memory and disk are handled by the operating system. Therefore, the programmer does not have to consider block replacement at all.

Furthermore, the ideal cache assumes full associativity even though the memory layers closest to the CPU typically are 2, 4, or 8-way set-associative (see Figure 2.2). The question is how efficiently operating systems are able to implement automatic replacement and full associativity?

According to Frigo et al. a fully associative LRU cache can be maintained in ordinary memory by letting data transfers be handled in software. By using hashing techniques, such an implementation can support access to any cache line in $O(1)$ expected time, using in total $O(M/B)$ records of size $O(B)$ words each. In the following we will explain how this is possible.

Consider the problem of distributing any subset S of a totally ordered universe of elements U evenly over a set of buckets L. L is much smaller than U. Using a 2-universal family of hashing functions [32] to distribute U over L, we can ensure that no matter which two distinct elements we map from U to L, the probability that they end up in the same bucket is the same. The consequence is, that no matter which subset S we choose, it will be expected to be evenly distributed over the buckets L.

In analogy to this description we have the following: Let M_{big} and M_{small} denote the sizes of two consecutive memory layers, and let B denote their block size. The two layers contain M_{big}/B and M_{small}/B blocks respectively. The smaller layer is x-way set-associative, so it is divided into M_{small}/x clusters containing x blocks each.

Now, M_{big} is a totally ordered universe of M_{big}/B blocks and M_{small}/x is a set of buckets. We can therefore use a 2-universal family of hashing functions to map an arbitrary subset of M_{small}/B blocks of the larger layer to the smaller layer. Doing so, we can expect the subset to be evenly distributed over the M_{small}/x buckets.

Within each cluster we can connect the blocks in a double-linked list to support the LRU policy. Due to the 2-universal hashing each cluster is expected to hold a constant number of blocks, namely x, so within a cluster the LRU policy is maintained in constant time. Since 2-universal hashing can be done in constant time the whole process is done within a constant time bound.

Cache-Oblivious Algorithms vs. External-Memory Algorithms

Since automatic replacement and full associativity can be handled in $O(1)$ time any optimal cache-oblivious algorithm can also be optimally implemented in the external-memory model (among others, Kumar noted this in [29]). To bypass the operating system, we can simply implement the 2-universal hashing technique in software to obtain automatic replacement between disk and main memory.

There is, however, still a difference between the optimal complexity bounds of some fundamental algorithms in the two models. Since it is up to the programmer to align data in the external-memory model, e.g., scanning an array of N elements incurs $\lceil N/B \rceil$ I/Os in that model, whereas the same task incurs $\lceil N/B \rceil + 1$ cache misses in the ideal-cache model.

Associativity and Memory Access Patterns

Though automatic replacement and full associativity are reasonable assumptions with respect to memory access times and space usage, full associativity may still cause problems. The memory access pattern of an algorithm may force a set-associative cache to only use one of its clusters. However, designing cache-oblivious algorithms is all about being true to the principle of locality of reference, so bad exploitation of set-associative caches is unlikely in this context.

3.4.4 Assumption: Tall-Cache

A last assumption of the ideal cache is that it is tall. The *tall-cache assumption* can be described as

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

meaning that the number of lines in the cache M/B is larger than the the block size B. Among others, Prokop [38] uses the tall-cache assumption in proving optimal cache complexity of matrix multiplication, matrix transposing and FFT. Sometimes the weaker assumption

$$M = \Omega(B^{1+\gamma})$$

suffices, where $\gamma > 0$. In the Lazy Funnelsort algorithm of Brodal and Fagerberg that we analyze in Chapter 6 this weaker assumption is used.

In very recent work Brodal and Fagerberg [14] emphasize the importance of the tall-cache assumption. In fact, they show that optimal comparison-based sorting is not possible without the tall-cache assumption. In contrast, the cache oblivious search tree that we analyze in Chapter 5 does not use the assumption at all.

Frigo et al. notes that in practice caches are usually tall. By a quick look at the cache characteristics of the Intel® Pentium® computer family (Figure 2.2) we are easily convinced that the tall-cache assumption is realistic.

3.4.5 The Constant Factors of the Ideal-Cache Model

The justifications of the ideal-cache model tells us that an algorithm that is *asymptotically* optimal in the ideal-cache model is also *asymptotically* optimal in any real memory system. While this is an important relation, it does not tell us precisely how much worse the cache complexity of an algorithm is on a real computer compared to the cache complexity in the ideal-cache model. As we have seen, the factor that separates these two complexities depends heavily on characteristics such as replacement policies and degrees of associativity, and in general we can therefore not determine its value. What we *do* know is that when we analyze algorithms in the ideal-cache model we can derive the constant factor of the most significant term and use it to estimate the *relative* performance of those algorithms. Therefore, this is what we will do in Chapter 5 and 6 when we analyze searching and sorting algorithms in the model.

3.5 Summary

In this chapter we have described the external-memory model, the hierarchical memory model, and the ideal-cache model. In the two former models knowledge of the memory system's characteristics is crucial in order to design memory efficient algorithms. In the ideal-cache model, no such knowledge is needed.

The ideal cache is theoretically justified, since it can be reduced into a cache that closer resembles real memory hierarchies, and that this reduction only increases the cache complexity of cache-oblivious algorithms by a constant factor.

CHAPTER 4

Analyzing Work Complexity

"There are only 10 types of people in the world. Those who understand binary and those who don't."
— unknown

The goal of *meticulous analysis* is to analyze the constant factors in the running time of algorithms. Especially, the constant in the most significant term in the function that describes the work complexity of an algorithm is of interest. In order to make precise predictions, meticulous analysis demands a computational model that closely matches the way real computers work. On the other hand, the model must be sufficiently simple, so the algorithms can be specified in a feasible way. In this chapter we will review two models suitable for meticulous analysis and decide on which model to use in our work complexity analysis.

4.1 The MMIX Model

The idea of analyzing constant factors in the running time of computer programs have, among others, been advocated by Knuth in his famous books on the art of computer programming (see, e.g., [25]).

Knuth's model of computation has until recently been the *MIX model*. As its name suggests the model is based on a mix of the most widely used computers of the 1960s and 1970s. The machine language of the MIX model is quite detailed. It defines about 100 instructions using multiple categories of registers, e.g. jump registers and index registers. No computer implementing the language has ever been built, so in a way the language is artificial.

Nevertheless, simulators and other tools for the MIX model exist for multiple platforms making it possible to actually write and execute programs written in the language.

As Knuth admits in the latest edition of his book ([25]) the MIX model is now quite obsolete, and he has therefore developed a new model, the *MMIX model* [27, 28]. The MMIX model builds on the same ideas as its predecessor, but being based on more recent 64-bit computers it now implements a RISC architecture. In both models each instruction is assigned a cost, and the running time of a program is the sum of the costs of the instructions executed. The cost varies among the instructions, so in the MIX model e.g. the `DIV` instruction always takes 12 time units whereas the `ADD` takes only 2. In the MMIX model the cost is further complicated by taking into account the number of memory references of an instruction and varying the cost of branch instructions based on whether the branch was predicted correctly.

Without doubt, the close connection to real computer architectures make the MMIX model suitable for very detailed algorithm analysis. This makes the model useful in book series such as Knuth's where the aim is not only to analyze running times of algorithms, but just as much to show how high-level language constructions are actually implemented in machines. However, the details of the model are overwhelming and specifying algorithms in the model is a tedious task. We need a simpler model.

4.2 Pure-C Cost Model

In Chapter 1 we presented the RAM model of computation. A variant of this is the *word-RAM* model of computation [20], which is also a model of computer machine languages. The word-RAM model differs from the traditional RAM model by restricting the contents of memory locations to be integers in the range $\{0, ..., 2^w - 1\}$, and that these integers are represented as strings of w bits. This allows for bitwise operations such as left and right shifts on integers. The word-RAM model is actually a family of related models differing in the arithmetic instruction set assumed to be available. Therefore, we can choose which set of instructions to support.

The pure-C language was first introduced by Katajainen & Träff [23], but in their analysis of mergesort programs they assumed the traditional RAM model. Bojesen et al. [11] assumed the word-RAM model, so they expanded the pure-C language with a few new operations matching word-RAM.

A combination of the word-RAM model and the pure-C language as defined by Bojesen et al. provides for a computational model less detailed than the MMIX model. We call this model the *pure-C cost model*. The model consists of a program, memory, a collection of registers, and a CPU to execute the program.

The pure-C cost model is machine independent, as the programs are written in the pure-C language, which is a subset of C [24]. All primitive operations in pure C have counterparts in the assembly languages of modern RISC computers.

4.2.1 The Pure-C Language

The fact that pure C is a subset of C makes it possible to compile pure-C programs into executables with any C compiler. As such, no special software is needed in order to carry out experiments and verifying the correctness of pure-C programs. This ease-of-use makes the pure-C cost model a lot more attractive than the MMIX model.

In the pure-C cost model memory locations and registers each contain one integer. Actual computations operate solely on registers and random access to memory locations are possible only by dereferencing registers. It should be mentioned that since the contents of registers are restricted to be within word size only $2^w - 1$ different memory locations can be referenced.

The pure C primitives are divided into 7 categories. In the following, `x` is a register (pointer or data), `y`, `z`, `s`, and `t` are registers (pointer or data) or constants, `p` is a pointer register, and λ is some label:

1. *Memory-to-register* and *register-to-memory* assignment statements.
   That is, the read statement `x = *p` and the write statement `*p = x` respectively.

2. *Register-to-register* and *constant-to-register* assignment statements.
   That is, `x = y`.

3. *Unary arithmetic expression* assignment statements.
   That is, `x =` $\ominus$ `y`, where $\ominus \in \{$`-`,`!`$\}$.

4. *Binary arithmetic expression* assignment statements.
   That is, `x = y` $\oplus$ `z`, where $\oplus \in \{$`+`,`-`,`*`,`/`,`&`,`|`,`<<`,`>>`$\}$.

5. *Conditional register-to-register* assignment statement.
   That is, `x = y` $\triangleleft$ `z ? s : t`, where $\triangleleft \in \{$`<`, `<=`, `==`, `!=`, `>=`, `>`$\}$.

6. *Conditional branch* statements.
   That is, `if (x` $\triangleleft$ `y) goto` λ, where $\triangleleft \in \{$`<`, `<=`, `==`, `!=`, `>=`, `>`$\}$, and λ is a label.

7. *Unconditional branch* statements.
   That is, `goto` λ, where λ is a label.

Normal C constructs such as functions and control statements (e.g., `switch` and `while`) are left out, though control statements can easily be described in Pure C, and functions can be assumed to be inlined.

4.2.2 Unit Cost

The original pure-C cost model was a unit-cost model. That is, the execution of each primitive was assumed to take the same amount of time τ, so a pure-C program of N instructions has a total cost of τN. However, in the latest revision of the model by Mortensen [31] the cost of *branch mispredictions* was introduced. In modern pipelined CPUs the execution of instruction B is already in process before the execution of the previous instruction A has completed. If instruction A is a conditional branch instruction, then we cannot be sure that instruction B is in fact the correct instruction to follow A, since we do not know whether the branch of instruction A should be taken when the execution of B begins. Modern CPUs use the technique of *speculative execution* to predict the outcome of conditional branch instructions, that is, the CPU guesses the outcome of the branch and speculatively begins to execute either the instructions following the branch or the instructions at the branch target. If the CPU does not guess the correct outcome of the branch, then a branch misprediction has occurred, and the instructions that were speculatively processed have to be flushed from the pipeline before program execution can resume. Therefore, a branch misprediction causes a delay in the pipeline. On the other hand, if the CPU predicts the branch correctly, then no special action is to be taken and the branch causes no execution time overhead.

As a rule of thumb Mortensen estimated the cost of the branch misprediction, denoted by τ_b, to be approximately 15τ[1].

Now, what remains in the model is a way to simulate the way in which the CPU guesses the outcome of a branch. Mortensen described two categories of conditional branches: Those that are easy to predict, and those that are hard to predict. To understand these categories consider the following code snip from [31]:

```
void limit_data(unsigned long* begin, unsigned long* end)
{
  unsigned long v;
  goto test;
loop:
  v = *begin;
  if (v < 100) goto skip_if;
  *begin = 100;
skip_if:
  begin = begin + 1;
test:
  if (begin < end) goto loop; /* hint: branch taken */
}
```

The program makes sure that no elements in the range [begin . . . end) are larger than 100.

[1]Mortensen derived this cost through experiments on both a 450 MHz Pentium® 2 and a 1000MHz AMD Athlon computer.

The conditional branch in line 12 is easy to predict, since it is taken for every iteration but the last. Mortensen states this by adding the comment `/* hint: branch taken */` to that line. Therefore, line 12 will incur only one branch misprediction.

On the other hand, the conditional branch in line 7 is hard to predict, since the computer does not know the values of the elements in the range [begin . . . end). In such a situation, Mortensen decided on simulating a branch prediction algorithm that guesses the outcome of the branch correctly every other time, i.e., in every other iteration. As a consequence, line 7 incurs $N/2$ branch mispredictions, and thus the total cost of the program becomes $5N\tau + (N/2 + 1)\tau_b$. As shown in the code snip no hints are given for conditional branches that are hard to predict.

Because branch mispredictions can be so expensive it is often advantageous to replace conditional branch instructions with conditional assignment instructions as they do not cause any branching. Mortensen illustrates how this is done by rewriting the `limit_data` program into the following code:

```
void limit_data(unsigned long* begin, unsigned long* end)
{
  unsigned long v;
  goto test;
loop:
  6 v = *begin;
  v = v > 100 ? 100 : v;
  *begin = v;
  begin = begin + 1;
test:
  if (begin < end) goto loop; /* hint: branch taken */
}
```

4.3 Summary

In this chapter we have reviewed two computational models suitable for meticulous analysis. The MMIX model is very closely connected to the way in which real computers work, but we feel that the details of the model are too overwhelming to make it usable for our purpose. The pure-C cost model is a simpler model. It has the advantage that programs written in the pure-C language can easily be compiled into fully executable programs by any C or C++ compiler. The pure-C model is a unit cost model that charges the cost τ to each instruction executed. In addition, the model takes branch mispredictions into account. Each branch misprediction is charged the cost of τ_b. In the following chapters we will use the pure-C cost model in our meticulous analyses[2].

[2]We will analyze the pure-C instruction count and the branch misprediction count separately. Therefore, we will leave out τ and τ_b.

CHAPTER 5

Static Search Trees

"There are rich counsels in the trees."

— Herbert P. Horne

According to Cormen et al. [17] search trees are data structures that support set operations such as searching for an element, extracting the minimum or maximum element, inserting elements and deleting elements. Search trees supporting operations that modify the tree, such as insert and delete, are *dynamic*, whereas search trees that do not allow this type of operations are *static*. It follows from this that static search trees have a limited scope, since they practically only support searching. The scope of dynamic search trees on the other hand is wider, since they can be used as priority queues, dictionaries, and similar data structures. However, the memory layout of cache-oblivious static search trees are relatively simple and used as building blocks of more complex cache-oblivious data structures.

In this chapter our main goal is to analyze the cache-oblivious static search tree, or more precisely, we investigate the task of searching in such a tree for a given element. We begin by taking a look at the previous work done on cache-oblivious search trees in Section 5.1. Thereafter, we present a formal description of search trees in Section 5.2. This leads us to present the idea of *layout policies* in Section 5.3, which can be seen as a general way of representing search trees both analytically and in practice. Layout policies for search trees offer a way of separating the memory layout of a tree from a generic algorithm used to perform the search.

To determine the efficiency of the cache-oblivious static search tree, we compare it to the well-known static binary search tree and a cache-aware static search tree. We analyze the binary search tree in Section 5.5, the cache-aware search tree in Section 5.6, and the cache-oblivious search tree in Section 5.7. Furthermore, we present an algorithm that builds a cache-oblivious search tree in $O(N)$ time in Section 5.7.3. This is an improvement of the $O(N \log_2 \log_2 N)$ algorithm of Ohashi [34], which, as far as we know, is the only algorithm for this task described in the literature so far.

For all search trees, our analysis covers both cache-efficiency analyzed in the ideal-cache model and the number of pure-C instructions performed for the various trees. Furthermore, to complete the analysis, we count the number of branch mispredictions caused by the trees. In Section 5.8 we sum up the theoretical results to get an idea of what we can expect from the benchmarks in Chapter 7.

5.1 Previous Work

The idea of cache-oblivious static binary search trees was already introduced in Prokop's thesis [38]. He shortly described the idea of a cache-oblivious memory layout with a cache complexity that is asymptotically equivalent to that of the B-tree [26], which is asymptotically optimal.

Independently, Ohashi [34] and Brodal et al. [15] described algorithms of asymptotically optimal work complexity for navigating in Prokop's memory layout, e.g., finding a child node from a parent. Brodal et al. furthermore compared the performance of the layout to other memory layouts and found the cache-oblivious approach able to compete with cache-aware layouts, and superior to memory layouts not explicitly optimized for the memory hierarchy. Ohashi used the layout for constructing a cache-oblivious heap.

Bender et al. [6] also used the idea of Prokop's cache-oblivious layout. Among other cache-oblivious data structures, they presented a dynamic search tree supporting search and update operations of optimal cache complexity $O(\log_B N)$ in the worst case. Their approach was mostly to use the tree as a building block for other structures, they did not analyze the tree in detail.

Ladner et al. [30] have recently made an experimental comparison of cache-aware and cache-oblivious static search trees using program instrumentation. They measured the execution time and by use of the instrumentation tool ATOM [40] they simulated the cache performance of various implementations. They found that the cache-oblivious implementations outperformed classic binary search on large datasets because of their better utilization of the cache layers of the memory system . Overall, they found cache-aware search to be the most effective.

As such, this chapter can be seen as a supplement to the experimental work of Ladner et al. [30]. However, as they remarked themselves, the cache-oblivious search algorithm they described was perhaps not the most efficient one known. We will analyze a simpler cache-oblivious search algorithm that is potentially more efficient.

5.2 Definitions

A binary search tree is a standard binary tree satisfying Property 1, stating the relative placement of the nodes in the tree [17].

Property 1. *Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y is less than or equal to x. If y is in the right subtree of x, then y is greater than or equal to x.*

The number of elements in each node of a search tree depends on the branching degree of the tree, i.e., a search tree with a branching degree of k has $k-1$ elements in each node. Therefore, Property 1 can be relaxed to consider branching degrees other than 2:

Property 2. *Let x be a node containing $k-1$ sorted elements in a search tree of branching degree k, where $k \geq 2$. Let $x_1, \ldots, x_{k-1}$ denote the non-decreasing sequence of elements in node x and let the k subtrees of a node be indexed $1, \ldots, k$ in a left to right manner. If y is a node in the left-most subtree of x (subtree 1), then all elements $y_1, \ldots, y_{k-1}$ of node y are less than or equal to x_1. If y is a node in the right-most subtree of x (subtree k), then all elements $y_1, \ldots, y_{k-1}$ of node y are greater than or equal to x_{k-1}. If y is a node in any other subtree i of node x (i.e., $2 \leq i \leq k-1$), then all elements $y_1, \ldots, y_{k-1}$ of node y are greater than or equal to x_i and less than or equal to x_{i+1}.*

For a static search tree of branching degree k to be *complete*, it is required that each node is either a leaf or has a branching degree of exactly k. It follows that the number of leaves in a complete static search tree is a power of k, and hence, the number of nodes is $k^i - 1$, where i is some positive integer. The *size* of a tree T is defined as the number of nodes in T, and should not be confused with the amount of memory occupied by T, which of course depends on both the number of elements in the tree and the size of these elements. The *height* h of a static search tree is the number of levels in the tree. If a complete static search tree T of degree k has N leaves, then $h(T)$ is $\log_k N + 1$. The *depth* of a node v is denoted $d(v)$ and is the number of nodes on the path from the root to v — including both the root and v.

5.3 Generic Static Search and Layout Policies

It follows from Property 2 that descending the levels in any search tree narrows down the interval to search by a factor k. So, no matter the branching degree of the tree, a search algorithm will have to examine the root for the element that is searched for, and, if the root does not contain that element, continue the search in one of its child nodes. If the element is found somewhere on the path from root to a leaf, then the algorithm returns `true`, otherwise `false`. What makes the search algorithm different for various search trees is the way the algorithm determines whether a node holds the element searched for and, if not, which child node to visit next and how to find that child node. How these matters are handled depends on both the branching degree and the way in which the tree is laid out in memory. We say, that the *layout policy* of the search tree describes these matters.

The algorithm `generic_search` (Program 2) defines an overall iterative search algorithm where the *LayoutPolicy* object hides the layout specific behavior. By examining the algorithm it is obvious, that in the worst case the loop will be executed once for each level of the tree.

```
template <typename RandomIterator, typename T, typename LayoutPolicy>
bool generic_search(RandomIterator begin,
                    RandomIterator beyond,
                    LayoutPolicy policy,
                    const T& value) {
  policy.initialize(begin, beyond);
  while(policy.not_finished()) {
    if (policy.node_contains(value)) {
      return true;
    }
    else
      policy.descend_tree(value);
  }
  return false;
}
```

Program 2: Generic static search.

The three search trees investigated in this chapter all follow the `generic_search` algorithm but have various layout policies. In our analysis the emphasis will be on the relative performance of these policies, so we will focus on how the various policies implement the functions `initialize()`, `not_finished()`, `node_contains()`, and `descend_tree()`.

5.4 Navigating in a Search Tree

There are two different ways of navigating in a search tree. Either we can use *explicit* or *pointer* navigation, where each parent node contains pointers to its children, or *implicit* navigation, where the positions of the child nodes of a parent node are calculated based on the position of the parent node in the tree. The use of explicit navigation means larger nodes and navigation involving few calculations, whereas implicit navigation means smaller nodes but navigation involving more calculations. Depending on the complexity of the explicit navigation procedure and possible constraints on how much space an algorithm is allowed to use, one navigation method may be more feasible than another.

5.5 The Inorder Layout Policy

We begin our investigation by examining the simplest of the layout policies, namely that of the static binary search tree.

The memory layout of the static binary search tree is straightforward. It is simply an array containing the elements in sorted order. This layout is sometimes called the *inorder* layout, because it resembles an inorder traversal of the tree. If the array is given by $A[0..N)$[1], then the root is the middle element with index $N/2$. If we denote this middle index by i, the indices of its left and right children can be calculated as $\lfloor i/2 \rfloor$ and $i + \lfloor i/2 \rfloor$ respectively.

Many different implementations of the binary search tree exist, so which one should we use? Mortensen [31] compared the performance of a number of different binary search programs. Among others, he benchmarked a lower bound algorithm similar to the one on which the SGI STL binary search is built[2]. He compared the running time of this lower bound algorithm to a number of optimized implementations. The benchmarks revealed that even though the optimized implementations were superior to the SGI STL implementation when data did fit into the level 1 and level 2 caches, the SGI STL implementation was most efficient for larger datasets. However, Mortensen's measurements were highly affected by the fact that he intentionally suppressed the compiler's ability to translate conditional branches into conditional assignments. This choice made the SGI STL implementation cause a high number of branch mispredictions and, by that, favored the optimized implementations that were explicitly programmed to use conditional assignments. For Mortensen, this was a necessary action to take since his goal was to measure the effects of using conditional branch instructions versus conditional assignment instructions in his programs.

Our goal is somewhat different. We are interested in deciding on whether we should use the SGI STL implementation as the base of our analysis or one of Mortensen's other implementations. Specifically, we are interested in knowing whether we should use conditional assignments in our programs instead of conditional branches that may cause mispredictions, and thereby decrease performance. To make the decision, we have compared the running

[1]We use $A[0..N)$ to emphasize that element 0 belongs to the A, while element N denotes the element one-past-the-end. This is similar to the use of *begin*() and *end*() in STL containers.

[2]The binary search implementation of SGI STL is both widely used and known to be quite efficient.

time of the SGI STL `lower_bound` to one of Mortensen's lower bound implementations that uses conditional assignments (see page 148 in Appendix A). Figure 5.1 shows that the effort of programming conditional assignments explicitly does not pay off in practice. Since the two implementation are equally efficient and the SGI STL version contains no conditional assignments, we decide on not making any workarounds to avoid conditional branches in our programs. We let the compiler perform this optimization job.

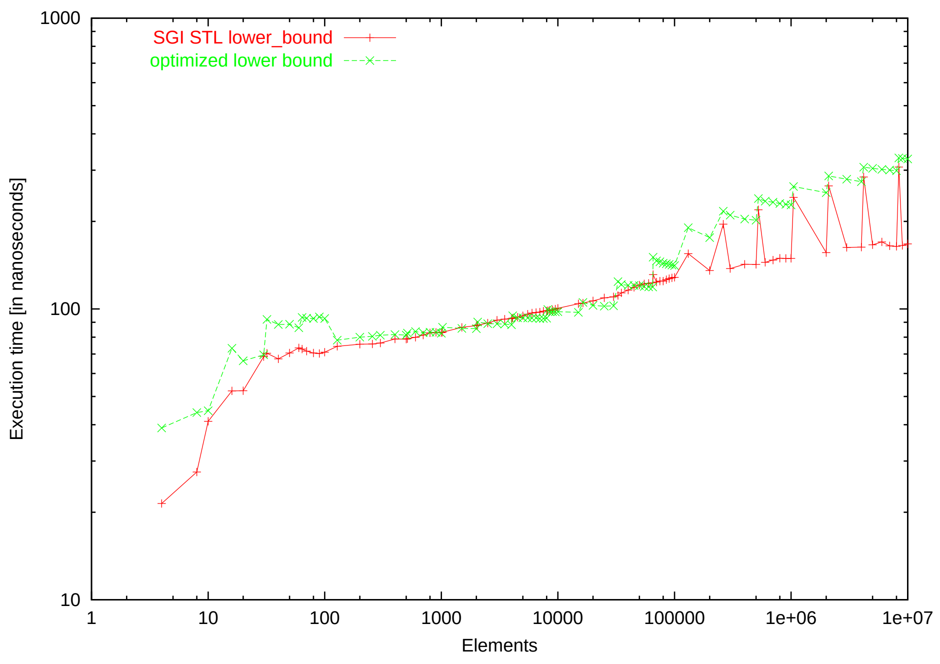


Figure 5.1: The running time of the SGI STL lower bound and an optimized lower bound implementation using conditional assignment instructions instead of conditional branches.

Based on the lower bound algorithm of SGI STL we have constructed an inorder layout policy to be used in the generic search algorithm of Program 2. This inorder policy is shown as Program 3. Figure 5.2 shows how the generic search algorithm works when it uses the inorder layout policy. In the figure, the search for the element 10 visits the elements 15, 7, 11, and 9 on the path from the root to element 10.

Work Complexity

We want to express the work complexity of the inorder layout policy by the number of pure-C instructions the generic search program executes when

```
template<typename RandomIterator, typename Value>
class inorder_search_policy {
  public:
    inorder_search_policy() { }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_begin = begin;
      m_beyond = beyond;
      m_len = beyond-begin;
    }
    inline bool not_finished() { return (m_len > 0); }
    inline bool node_contains(const Value& value) {
      m_half = m_len >> 1;
      m_middle = m_begin;
      m_middle = m_middle + m_half;
      return *m_middle == value;
    }
    inline void descend_tree(const Value& value) {
      if (*m_middle < value) {
        m_begin = m_middle;
        ++m_begin;
        m_len = m_len - m_half - 1;
      }
      else
        m_len = m_half;
    }
  private:
    typename std::iterator_traits<RandomIterator>::difference_type m_half;
    typename std::iterator_traits<RandomIterator>::difference_type m_len;
    RandomIterator m_begin, m_beyond, m_middle;
};
```

Program 3: Layout policy of binary search.

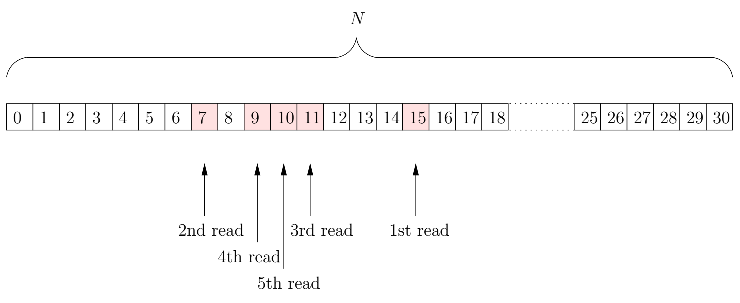


Figure 5.2: The nodes of a compete binary search tree are laid out in memory in sorted order, resembling an inorder traversal of the tree. When searching for element 10 the inorder layout policy visits nodes 15, 7, 11, 9, and 10 in the stated order.

using that particular policy. Program 4 is a pure-C translation of the generic search program using the inorder layout policy[3].

```
template <typename RandomIterator, typename T>
bool inorder_search(RandomIterator begin,
                    RandomIterator beyond,
                    const T& value) {
   typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
 initialize:
   diff_type half, len = beyond - begin;
   T middle_value;
   RandomIterator middle;

   goto not_finished;
 descend_right:
   begin = middle;
   begin = begin + 1;
   len = len - half;
   len = len - 1;
 not_finished:
   if (len == 0) goto return_false; /* hint: not taken */
 node_contains:
   half = len >> 1;
   middle = begin + half;
   middle_value = *middle;
   if (middle_value == value) goto return_true;
   if (middle_value < value) goto descend_right;
 descend_left:
   len = half;
   goto not_finished;

 return_false:
   return false;
 return_true:
   return true;
}
```

Program 4: Pure-C translation of generic search using the inorder layout policy.

Theorem 2. *Searching in a complete binary tree laid out in memory according to the inorder layout incurs at most* $10\lfloor \log_2 N \rfloor + O(1)$ *Pure-C instructions, if using the generic search algorithm as defined in Program 2.*

Proof. From the generic search algorithm it is apparent that the while loop will dominate the running time. Therefore, we are interested in analyzing

[3]The pure-C programs are written in STL style. Therefore, variables of the type *RandomIterator* are expected to behave like random access iterators as defined in the C++ standard [22].

the pure-C instructions of this loop, i.e., lines 11–27 of Program 4. Lines 6–9 and 29–32 will only add a constant $O(1)$ term to the running time.

There are two ways of leaving the while loop. Either the value searched for is found in line 23 or repetitive executions of the loop will have truncated the length of the sequence to 0 in line 18. The latter situation will happen in the worst case, so the question is how many times the loop is executed before `len` becomes zero.

Descending down the tree from a parent to its left child cuts the sequence N into a sequence of length $\lfloor N/2 \rfloor$, while descending down to its right child cuts the sequence N into a sequence of length $N - \lfloor N/2 \rfloor - 1$. Since $\lfloor N/2 \rfloor \geq N - \lfloor N/2 \rfloor - 1$, a search that continuously descends down the left path makes up the worst-case search.

Let $T(N)$ define the number of times the loop is executed. If N is 0, then the loop is not executed at all, so we have $T(0) = 0$. If $N > 0$, then $T(N)$ is bounded by the recurrence $T(N) \leq T(\lfloor N/2 \rfloor) + 1$. Assuming that $T(\lfloor N/2 \rfloor) \leq \lfloor \log_2 N \rfloor + 1$ yields

$$\begin{aligned} T(N) &\leq T(\lfloor N/2 \rfloor) + 1 \\ &\leq \lfloor \log_2 \lfloor N/2 \rfloor \rfloor + 2 \\ &\leq \lfloor \log_2 N/2 \rfloor + 2 \\ &= \lfloor \log_2 N - 1 \rfloor + 2 \\ &= \lfloor \log_2 N \rfloor + 1. \end{aligned}$$

This bound also holds for the base case $T(1)$.

Descending down the rightmost path of the tree incurs 10 pure-C instructions per loop iteration, hence we have a worst case pure-C complexity of $10\lfloor \log_2 N \rfloor + O(1)$. □

Branch Mispredictions

The following theorem states the number of branch mispredictions incurred by the inorder layout:

Theorem 3. *Searching in a complete binary tree laid out in memory according to the inorder layout incurs at most $\lfloor \log_2 N \rfloor + 2$ branch mispredictions, if using the generic search algorithm as defined in Program 2.*

Proof. To derive the branch misprediction count we consider the conditional branch instructions in line 18, 23, and 24 of Program 4. The branch in line 18 has a hint, so it will only cause a misprediction in the last iteration of the loop. None of the branches in line 23 and 24 have hints, so they are predicted correctly every second iteration, i.e., $\frac{\lfloor \log_2 N \rfloor + 1}{2}$ times each. Hence, the inorder layout incurs $\lfloor \log_2 N \rfloor + 2$ branch mispredictions in total. □

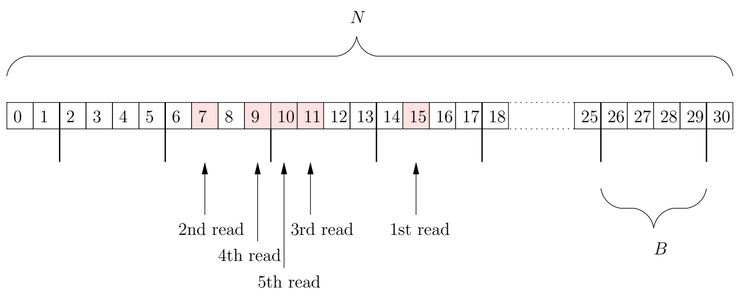


Figure 5.3: As on Figure 5.2, the search for element 10 descends through nodes 15, 7, 11, 9, and 10. During the search 3 distinct cache blocks are visited.

Cache Complexity

To derive the cache complexity of binary search incurred by the inorder layout, consider the example on Figure 5.3. The example shows how the iterations narrow down the sequence to search, and by that make the memory accesses increasingly more local. Assuming that a single element will never cross a boundary between two blocks, Theorem 4 states the number of cache misses incurred by the search algorithm.

Theorem 4. *Searching in a binary tree, laid out in memory according to the inorder layout, requires at most* $Q(N) = \lceil \log_2 N \rceil - \lfloor \log_2 B \rfloor + 2$ *cache misses, where* N *is* $2^i - 1$ *for some positive integer* i.

Proof. Searching the tree involves, in the worst case, traversing a path from the root all the way to a leaf. The number of cache misses involved in that process is given by the recurrence $T(N) = T(N/2)+1$. For each node visited, the problem size is reduced by half, and visiting a node requires reading a new block into the cache. The recursion repeats until a subtree fits in a cache line, i.e., the size of a subtree is less than B. This means that the base case of the recurrence is $T(B) = 2$. For each of the $\log_2 N$ levels in the tree a cache miss may occur, except for the final $\lfloor \log_2 B \rfloor$ levels, that incur only two misses; one miss for reading the block, and one additional miss if the block alignment does not fit. Therefore, the total number of cache misses becomes $\lceil \log_2 N \rceil - \lfloor \log_2 B \rfloor + 2$. Rounding down $\log_2 B$ is necessary, since, when the recursion stops, we can only be sure that the subtree has at most size B, not exactly size B. □

The constraint in Theorem 4, that N has to be of the form $2^i - 1$, can easily be relaxed by using a tree that is a little bigger than actually needed, i.e. of size $2^{\lceil \log_2 N \rceil} - 1$, and allowing empty nodes and leaves in the tree. The size of such a tree will be at most $2N - 1$ resulting in a worst-case space overhead of size $N-1$. In fact, due to the rounding up of $\log_2 N$ in Theorem 4, the upper bound stated by the theorem holds for all values of N.

As revealed by the analysis, the search algorithm is only likely to make cache hits when a subtree becomes sufficiently small. Therefore, it can be argued, that the inorder layout makes searching incompatible with the principle of spatial locality.

5.6 The Cache-Aware Layout Policy

In the inorder layout, all but the last $\lfloor \log_2 B \rfloor$ descends from a parent to a child node incurs a cache miss. The cache-aware layout policy that we present in this section does a better job at keeping data accesses local. The layout is based on a description from Bentley [9]. He calls the searching algorithm for this layout the *multiway heap search*, so we will use the term *heap policy* for the search policy implementing this cache-aware memory layout.

The heap policy uses multiway branching in order to obtain good cache complexity. This approach ensures a tree with fewer levels than the inorder layout, and since we are allowed to use knowledge of the cache line length, we can ensure that cache misses do not occur as long as we stay inside a node. Only when we descend a level down the tree it may incur a cache miss. This way of laying out data in accordance with cache characteristics is sometimes called *blocking*.

5.6.1 Implicit Navigation

Assuming 32-byte cache lines, 4-byte elements, and implicit navigation, each node in the heap layout can contain 8 elements. Hence, we use 9-way branching, so for less than 9 elements the tree will have 1 level, for 9–80 elements the tree will have 2 levels and so forth. We only need 11 levels to contain 2^{32} elements. Within a node the elements are stored inorder, so searching inside a node is done by use of standard binary search. The concept of the implicit heap layout is shown on Figure 5.4.

According to Bentley, the heap layout uses B-way branching instead of the $(B+1)$-way branching shown on the figure. Therefore, Bentley's nodes contain perfectly balanced binary trees of height $\log_B N$, but in each cache block a few bytes will remain unused. On the contrary, using $(B+1)$-way branching means fully filled cache blocks, but unbalanced binary trees of height $\log_{B+1} N$. While this difference is a minor detail, filling the cache blocks completely means a small improvement in cache performance.

Beginning at the root, the implicit heap search works as follows. First, we check if the value we search for is held within the root. This can be

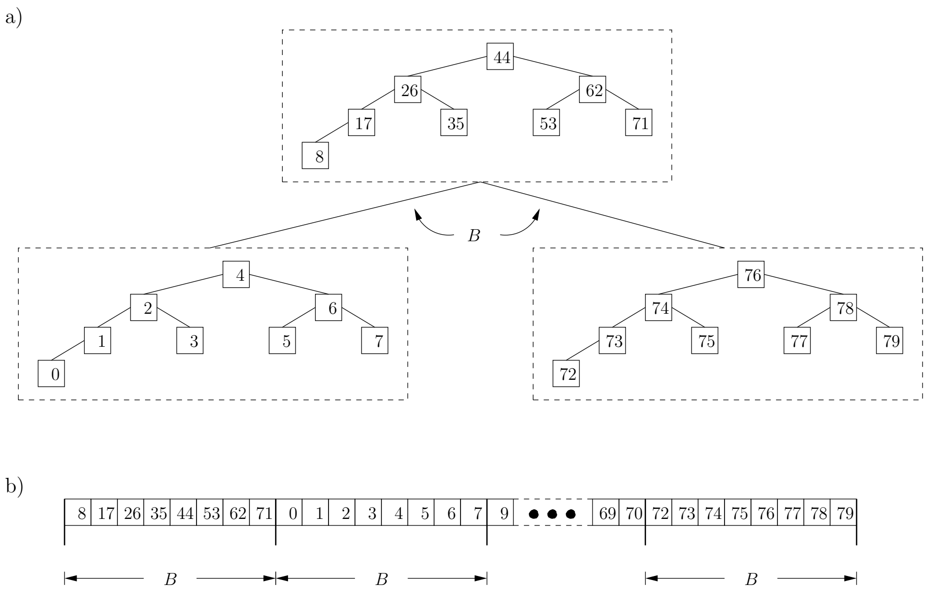


Figure 5.4: A tree of 2 levels in the implicit heap layout. a) The implicit heap layout uses $(B+1)$-way branching with B elements in each node. b) The B elements of a node are placed continuously in memory.

accomplished by use of the STL `lower_bound` algorithm[4], as the elements within a node are sorted in increasing order. If the lower bound algorithm returns the value we are looking for, then we are done. Otherwise the lower bound algorithm will indicate in which child node the search should proceed by returning the index of the smallest element larger than the one we are looking for. We can now calculate the memory position of the child to visit next. The nodes are stored levelwise in memory, that is, the root at depth 1 is placed prior to the nodes at depth 2, and so on. Therefore, assuming that we know the index i and the address i_a of the node we are currently in (i.e., the index points to the first element in the node), and the address l_a of the node returned by the lower bound algorithm, then the index of the child to visit next is calculated as

$$i = i * (B+1) + B * ((l_a - i_a) + 1),$$

where the $i * (B+1)$ term brings the search down one level in the tree, and the $B * ((l_a - i_a) + 1)$ term makes sure that we index the correct node at that level.

As an example, consider searching for the element 5 in the tree on Figure 5.4. The root has index 0, and the lower bound algorithm will return the

[4]We might as well use the upper bound algorithm, as the algorithms exhibit identical behavior for unique keys.

address of element 8, so the node to visit next is at index $0 * (8 + 1) + 8 * (0 + 1) = 8$.

Work Complexity

The implicit heap layout can be expressed as a layout policy (Program 5) and used in the `generic_search` algorithm. Program 6 is a full translation of the pure-C program resulting from combining the `generic_search` algorithm and this layout. The following theorem states the work complexity of searching in the layout.

```
template <typename RandomIterator, typename Value>
class implicit_heap_policy {
  public:
    implicit_heap_policy(int degree) { m_degree = degree; }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_index = 0;
      m_size = beyond-begin-m_degree;
      m_current_node = begin;
      m_lower_bound = begin;
      m_begin = begin;
    }
    inline bool not_finished() { return m_index < m_size; }
    inline bool node_contains(const Value &value) {
      m_lower_bound =
        std::lower_bound(m_current_node, m_current_node+m_degree-1, value);
      return (*m_lower_bound == value);
    }
    inline void descend_tree(const Value& value) {
      m_index =
        m_index*m_degree + (m_degree-1)*((m_lower_bound-m_current_node)+1);
      m_current_node = m_begin+m_index;
    }
  private:
    int m_degree;
    typename iterator_traits<RandomIterator>::difference_type
      m_index, m_size;
    RandomIterator m_current_node, m_lower_bound, m_begin;
};
```

Program 5: The implicit heap policy.

Theorem 5. *Searching in an array of N elements laid out in memory according to the implicit heap layout incurs at most $(13+9\log_2 B)\lfloor \log_{(B+1)} N \rfloor + O(1)$ pure-C instructions, if using the generic search algorithm as defined in Program 2.*

Proof. As was the case with the inorder policy, the while loop of `generic_search` dominates, so our focus is on lines 13–27 of Program 6. In the worst

```
template <typename RandomIterator, typename T>
bool implicit_heap_search(RandomIterator begin,
                          RandomIterator beyond,
                          int degree,
                          const T& value) {
  typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
 initialize:
  RandomIterator lower_bound, l_middle, current_node = begin;
  diff_type l_len, l_half, index = 0, size = beyond-begin;
  T l_middle_value, x;
  int degree_minus_1 = degree - 1;

  goto not_finished;
 descend_tree:
  index = index * degree;
  x = lower_bound-current_node;
  x = x + 1;
  x = degree_minus_1 * x;
  index  = index + x;
  current_node = begin + index;
 not_finished:
  if (index >= size) goto return_false; /* hint: not taken */
 node_contains:
  lower_bound = current_node;
  LOWER_BOUND(lower_bound, x)
  if (x == value) goto return_true;
  goto descend_tree;

 return_false:
  return false;
 return_true:
  return true;
}
```

Program 6: The pure-C implementation of the implicit heap search.

case the while loop is exited in line 22 as the consequence of an unsuccessful search. So, how many times is the loop executed before `index` becomes larger than or equal to `size`?

The nodes in the tree are laid out levelwise in memory, so descending down one level in the tree from depth d_i to depth d_{i+1} corresponds to having moved at least $(B+1)^i - 1$ places to the right in the array since the search began. Therefore, `index` will become larger than or equal to `size` when we have descended $\lfloor \log_{(B+1)} N \rfloor$ levels in the tree. As the loop is executed once before the first descend, it is executed $1 + \lfloor \log_{(B+1)} N \rfloor$ times in total.

Without counting the lower bound calculation of line 25, the loop consists of 10 pure-C instructions. The lower bound can be calculated by Program 4 with a slight modification. By excluding line 23 of that program, and returning the iterator `begin` instead of true or false, the program implements the lower bound algorithm. The lower bound program as used in Program 6 is shown in Program 7. It executes $9\lfloor \log_2 B \rfloor + 3$ pure-C instructions (cf. Theorem 2).

```
#define LOWER_BOUND(lower_bound, x) \
   l_len = degree_minus_1; \
   goto l_not_finished; \
  l_right: \
   lower_bound = l_middle; \
   lower_bound = lower_bound + 1; \
   l_len = l_len - l_half; \
   l_len = l_len - 1; \
  l_not_finished: \
   if (l_len == 0) goto l_return; /* hint: not taken */ \
  l_found: \
   l_half = l_len >> 1; \
   l_middle = lower_bound + l_half; \
   l_middle_value = *l_middle; \
   if (l_middle_value < value) goto l_right; \
  l_left: \
   l_len = l_half; \
   goto l_not_finished; \
  l_return: \
   x = *lower_bound;
```

Program 7: Lower bound calculation in pure-C.

Since B in practice is a power of 2, we can omit the floors of the lower bound calculation, so the worst-case pure-C complexity becomes $(13 + 9\log_2 B)\lfloor \log_{(B+1)} N \rfloor + O(1)$, $\square$

Branch Mispredictions

Theorem 6. *Searching in an array of N elements laid out in memory according to the implicit heap layout incurs at most*

$$\tfrac{1}{2}\left(\lfloor\log_{(B+1)} N\rfloor + 1\right)(\log_2 B + 4) + 1$$

branch mispredictions, if using the generic search algorithm as defined in Program 2.

Proof. The loop of Program 6 (line 13–27) contains two conditional branch instructions plus the conditional branch instructions of the lower bound calculation in line 25. The branch in line 22 is only mispredicted in the last iteration, while the branch in line 26 is mispredicted every second iteration of the loop, that is $\frac{1+\lfloor\log_{(B+1)} N\rfloor}{2}$ times. One lower bound calculation (Program 7) involves two conditional branch instructions in line 10 and 15. The branch in line 10 is mispredicted only once, while the branch in line 15 is mispredicted $\frac{1+\log_2 B}{2}$ times. Hence, in total Program 6 causes

$$\begin{aligned}
& 1 + \frac{1+\lfloor\log_{(B+1)} N\rfloor}{2} + \left(1 + \lfloor\log_{(B+1)} N\rfloor\right)\left(1 + \frac{1+\log_2 B}{2}\right) \\
= \; & 1 + \frac{1}{2} + \frac{\lfloor\log_{(B+1)} N\rfloor}{2} + \left(1 + \lfloor\log_{(B+1)} N\rfloor\right)\left(1 + \frac{1}{2} + \frac{\log_2 B}{2}\right) \\
= \; & \frac{3}{2} + \frac{\lfloor\log_{(B+1)} N\rfloor}{2} + \frac{3}{2} + \frac{\log_2 B}{2} + \frac{3\lfloor\log_{(B+1)} N\rfloor}{2} + \frac{\lfloor\log_{(B+1)} N\rfloor\log_2 B}{2} \\
= \; & \frac{4\lfloor\log_{(B+1)} N\rfloor + \log_2 B + \lfloor\log_{(B+1)} N\rfloor\log_2 B + 6}{2} \\
= \; & \frac{1}{2}\left(\lfloor\log_{(B+1)} N\rfloor + 1\right)(\log_2 B + 4) + 1
\end{aligned}$$

branch mispredictions. □

Cache Complexity

In a cache-aware algorithm we are allowed to take advantage of knowing the cache-line length, so we can avoid the alignment considerations mentioned in Section 3.3.3 and assume that the nodes are aligned on cache-line boundaries. The following theorem states the cache complexity of using the implicit heap layout.

Theorem 7. *Searching in an array of N elements laid out in memory according to the implicit heap layout incurs at most $Q(N) = \lceil\log_{(B+1)} N\rceil$ cache misses.*

Proof. The search visits one node at each level of the tree. As all nodes are aligned on cache line boundaries and take up exactly one cache line each, the cache complexity equals the number of levels in the tree, which is given by $\lceil\log_{(B+1)} N\rceil$. □

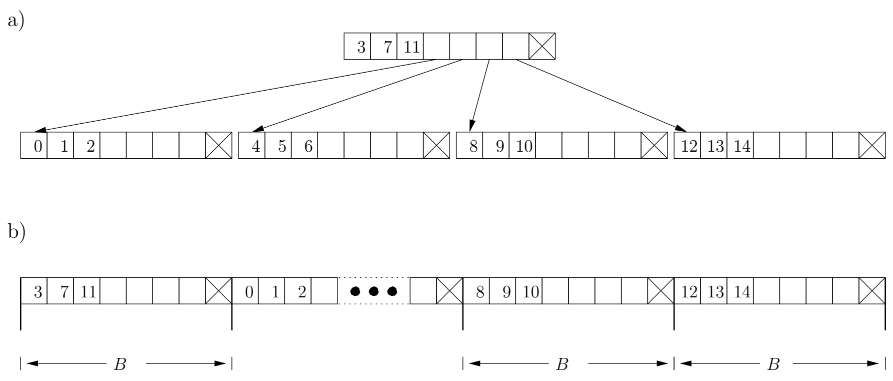


Figure 5.5: A tree of 2 levels in the explicit heap layout. In the example the cache line length is 32 bytes and each element takes up 4 bytes. **a**) The explicit heap layout uses $(B/2)$-way branching with $(B/2) - 1$ elements in each node. **b**) The nodes are placed levelwise and continuously in memory.

5.6.2 Explicit Navigation

The use of explicit or pointer navigation instead of implicit navigation offers a trade-off between the work and cache complexity of the multiway heap search. By simplifying the navigational computations explicit navigation can lower the constant in the leading term of the work complexity. The price to pay is an increase in the logarithmic bases of both the work and the cache complexities.

Assuming 32-byte cache lines, 4-byte elements, and 4-byte pointers[5], the explicit navigation leaves only room for three elements in each node. Of the remaining 20 bytes, 16 bytes contain four 4-byte pointers to the nodes in the next layer, and the last 4 bytes are unused. The concept of the explicit heap layout is depicted on Figure 5.5.

The search algorithm still determines which child node to visit next by calculating the lower bound in the parent node. But instead of calculating the position of the child nodes in memory, the position is found by following a pointer.

Work Complexity

Program 8 depicts the explicit heap policy and Program 9 is the pure-C translation. In Program 8 and 9 a node is expected to be a C `struct` containing an array of $B/2 - 1$ elements and an array of $B/2$ pointers to the children of that node. In leaf nodes, the child pointers are expected to point to `beyond`. The structure of the nodes is shown in Figure 5.5.

[5]32 bits per pointer is reasonable to assume as long as we use 32-bit computers.

```
template <typename RandomIterator, typename Value>
class explicit_heap_policy {
  public:
    explicit_heap_policy(int degree) { m_degree = degree; }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_current_node = begin;
      m_beyond = beyond;
    }
    inline bool not_finished() {
      return (m_current_node < m_beyond);
    }
    inline bool node_contains(const Value &value){
      m_lower_bound =
        lower_bound(&(m_current_node->e[0]), &(m_current_node->e[m_degree-1]), value);
      return (*m_lower_bound == value);
    }
    inline void descend_tree(const Value& element) {
      m_current_node =
        m_current_node->p[m_lower_bound-reinterpret_cast<Value*>(m_current_node)];
    }
  private:
    int m_degree;
    RandomIterator m_current_node, m_beyond;
    Value *m_lower_bound;
};
```

Program 8: The explicit heap policy.

```
template <typename RandomIterator, typename T>
bool explicit_heap_search(RandomIterator begin,
                          RandomIterator beyond,
                          unsigned int degree_minus_1,
                          const T& value) {
  typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
 initialize:
  T x, l_middle_value, *lower_bound, *l_middle;
  diff_type y, l_len, l_half;
  RandomIterator current_node = begin;

  goto not_finished;
 descend_tree:
  y = lower_bound - reinterpret_cast<T*>(current_node);
  current_node = current_node->p[y];
 not_finished:
  if (current_node >= beyond) goto return_false; /* hint: not taken */
 node_contains:
  lower_bound = reinterpret_cast<T*>(current_node);
  LOWER_BOUND(lower_bound, x)
  if (x == value) goto return_true;
  goto descend_tree;

 return_false:
  return false;
 return_true:
  return true;
}
```

Program 9: The pure-C implementation of the explicit heap search.

a)

```
typedef struct a {
  unsigned int b[3];
  unsigned int *c[4];
} a;
a *p;
unsigned int *x;
x = p->c[2];
```

b)

```
movl    p, %eax
movl    20(%eax), %eax
movl    %eax, x
```

Figure 5.6: The compiler translates the last line of program a) into assembly code b), which corresponds to three pure-C instructions, namely a memory-to-register, a register-to-register, and a register-to-memory instruction. Assuming that p is already in memory prior to the indexing of $c[2]$, only the cost of the last two assembly instructions should be charged to the struct indexing.

Pure-C has no instructions for accessing members of a `struct` (line 15 in Program 9), so we have to determine the cost of such an instruction to be able to express the program's work complexity in the pure-C cost model. By use of Program 5.6a and the `g++` compiler with the `-S` option we can determine this cost. The compiler translates the line `x = p->c[2]` into the three assembly instructions of Program 5.6b. Since `c[2]` is found by adding 20 to the pointer `p`, the assembly instructions tell us that the members of a `struct` are byte-indexed on basis of the address of the `struct` itself. In total, `x p->c[2]` corresponds to a memory-to-register, a register-to-register, and a register-to-memory instruction. Assuming that p is kept in a register prior to the access, we charge it a pure-C cost of 2. This resembles the way in which the cost of other pure-C instructions are derived. E.g., the instruction `x = x * 2` only has a pure-C cost of 1, since `x` is assumed to be in a register prior to the multiplication.

Theorem 8. *Searching among N elements laid out in memory according to the explicit heap layout incurs at most $(9\log_2(B/2-1)+7)\lfloor\log_{(B/2)} N\rfloor + O(1)$ pure-C instructions, if using the generic search algorithm as defined in Program 2.*

Proof. We focus on the loop in the lines 12–22 of Program 9. In the worst case the loop is exited in line 17 as the consequence of an unsuccessful search, that is, when the iterator `current_node` becomes larger than or equal to the iterator `beyond`.

The nodes in the tree are laid out in memory levelwise. Therefore, descending down one level in the tree from depth d_i to depth d_{i+1}, by following a child pointer (line 15), corresponds to increasing the value of `current_node`, so that all nodes at level $j < i$ are at lower addresses than that node. Therefore, `current_node` is larger than or equal to `beyond` after $1 + \lfloor \log_{(B/2)} N \rfloor$ iterations.

Including the calculation of lower bound in line 20, the work complexity becomes $(9 \log_2(B/2 - 1) + 7)\lfloor \log_{(B/2)} N \rfloor + O(1)$. □

Branch Mispredictions

Theorem 9. *Searching in an array of N elements laid out in memory according to the explicit heap layout incurs at most*

$$\tfrac{1}{2}\left(\lfloor \log_{(B/2)} N \rfloor + 1\right)(\log_2(B/2 - 1) + 4) + 1$$

branch mispredictions, if using the generic search algorithm as defined in Program 2.

Proof. Program 9 contains conditional branch instructions similar to those of Program 6, i.e., the two conditional branch instructions in line 17 and 21 plus the conditional branch instructions of the lower bound calculation in line 20. Therefore, the only difference between the branch misprediction counts of the implicit and the explicit heap layout is the logarithm base and the number of elements that the lower bound algorithm works on in each iteration. Hence, Program 9 causes the stated number of branch mispredictions. □

Cache Complexity

The cache complexity of using the explicit heap layout resembles that of its implicit sibling. The only difference is the branching degree.

Theorem 10. *Searching in an array of N elements laid out in memory according to the explicit heap layout incurs at most $Q(N) = \lceil \log_{(B/2)} N \rceil$ cache misses.*

Proof. The search visits one node at each level of the tree. As all nodes are aligned on cache line boundaries and take up less than one cache line each, the cache complexity equals the number of levels in the tree, which is given by $\lceil \log_{(B/2)} N \rceil$. □

5.7 The Cache-Oblivious Layout Policy

The cache-oblivious layout policy that we present in this section offers a cache complexity that is comparable to that of the cache-aware layout, but without using any knowledge of the memory system's characteristics. The layout is sometimes referred to as the *van Emde Boas layout*, since it matches the layout of a priority queue of van Emde Boas. We prefer to use the term

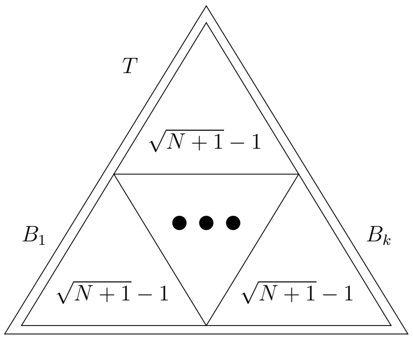


Figure 5.7: The concept of splitting a tree in the height partitioned layout. The tree with N nodes is split at the middle level resulting in a top recursive subtree T and $\sqrt{N+1}$ bottom recursive subtrees $B_1, B_2, \ldots, B_k$. Each subtree has size $\sqrt{N+1}-1$.

height partitioned layout, since it better describes what the layout is all about.

The idea of the height partitioned layout is as follows. Suppose we have a complete binary tree of height h with N nodes storing N elements in search-tree order, and that h is a power of 2. The tree is split at the middle level of edges, i.e., between nodes at depth $h/2$ and $h/2+1$ resulting in a top recursive subtree T of height $h/2$ and a number of disjoint bottom recursive subtrees $B_1, \ldots, B_k$ each of height $h/2$. The layout is obtained by recursively laying out the top subtree T and the bottom subtrees $B_1, \ldots, B_k$ in memory in the order $T, B_1, \ldots, B_k$. We now have $\sqrt{N+1}$ bottom subtrees in addition to the top subtree, and since the tree is complete, each subtree is of size $\sqrt{N+1}-1$.

If h is not a power of two, then splitting will not result in top and bottom subtrees of the same height. Instead, the tree is split so that the height of the bottom trees is $2^{\lceil \log_2 (h/2) \rceil}$, i.e., the smallest power of 2 greater than or equal to $h/2$. This leaves the top recursive subtree with height $h - 2^{\lceil \log_2 (h/2) \rceil}$. This splitting method continues on the top recursive subtree until its height becomes a power of 2. Notice, that if there still is an "odd-sized" subtree when the recursion stops, then this splitting method guarantees that it is located at the topmost top subtree. Furthermore, notice that all subtrees except a possibly "odd-sized" top subtree at this level of recursion all have the same size.

Figure 5.7 illustrates the concept of splitting a tree with the height partitioned layout and Figure 5.8 is an example of the height partitioned layout for a tree of height 5.

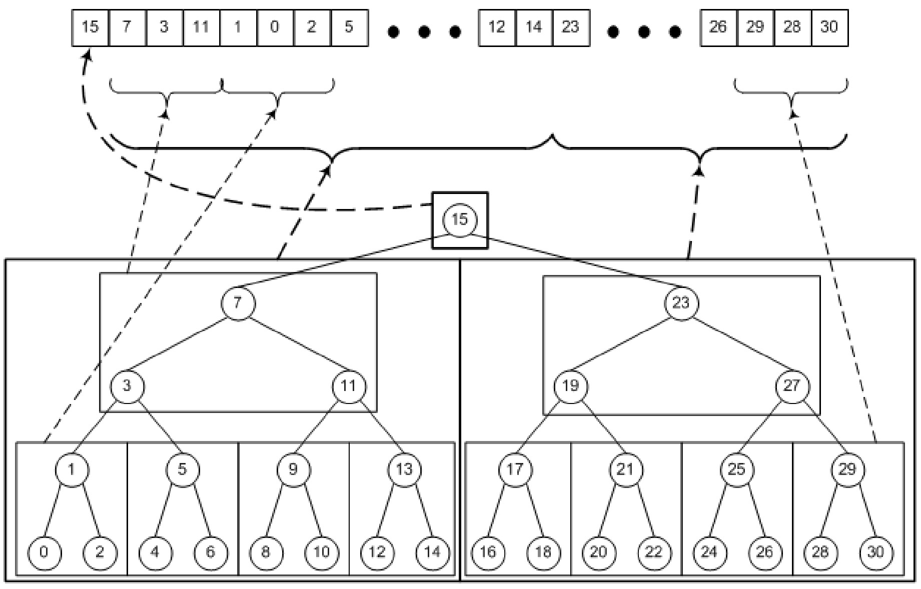


Figure 5.8: The height partitioned layout of a tree of height 5. Since 5 is not a power of 2, the tree is split so that the bottom recursive subtrees have heights that are powers of 2, namely 4. This leaves the top subtree with height $5 - 4 = 1$. As can be seen the top subtree is recursively laid out in memory, followed by the bottom subtrees — here in a left to right order. Actually the order is not important as long as a subtree is laid out in a continuous segment of memory.

5.7.1 Implicit Navigation

As was the case with the inorder layout, navigating down the height partitioned tree implicitly involves calculating the location of left and right child nodes given the location of their parent. In the inorder layout implicit navigation involved simple address arithmetics. In the cache-oblivious layout it gets more complicated. A clever implicit navigation method of Brodal et al. [15] is described in this section.

The navigation method builds on two observations. The first is on the height partitioned layout, and the second is on the breadth-first layout[6] of a tree of the same size:

Observation 1: At some level in the recursion, every node will become the root of a bottom tree. If we unroll the recursion until this is true for a node v, then the unrolling will be just the same for all other nodes at depth $d(v)$. Hence, the bottom subtrees with root at level $d(v)$ all have the same size, and they all have corresponding top subtrees of the same size. In Figure 5.9a, the first level of recursion splits the tree at depth 2, the second at depth 4, and the third at depth 3 and 5. Hence, the split at depth 5 makes the nodes 3, 4, 18, and 19 roots of bottom trees of size 1 after three levels of recursion. The corresponding top subtree of node 3 and 4 is rooted at node 2 and has size 1 — the same size as node 17, which is the corresponding top subtree of node 18 and 19.

Observation 2: Given a node v at position i in a tree of the breadth-first layout, the positions of its children are given by $2i$ and $2i+1$. Figure 5.9b depicts a tree of the breadth-first layout. A tree of the breadth-first layout holds the following property:

Property 3. *Let i_{bin} be the sequence of bits representing the position of a node v in the breadth-first layout. The bits of i_{bin}, except for the most significant one, represent by descending significance the left and right turns on the path from the root the v. A 0-bit represents a left turn and a 1-bit represents a right turn.*

In Figure 5.9b, the binary representation of node 25 is 11001_{bin}. Therefore, the path from the root to node 25 is right, left, left, right.

By observation 1, and by recalling that the height partitioned layout is all about storing bottom subtrees in memory after their corresponding top subtree (Figure 5.8), the memory location of the bottom subtrees $B^a_1, \ldots, B^a_k$ of a corresponding top subtree T^a, relative to the root of T^a, is the same as the memory location of the bottom subtrees $B^b_1, \ldots, B^b_k$ of a corresponding

[6]The breadth-first layout is similar to the layout of a binary heap.

top subtree T^b, relative to the root of T^b, if the roots of T^a and T^b are at the same depth. As an example, in Figure 5.9, the bottom subtree rooted at node 20 is the left-most bottom subtree of the top subtree rooted at node 17. The distance of 3 memory units is the same as the distance between the left-most bottom subtree rooted at node 5 and the top subtree rooted at node 2. This means that we are able to calculate the position of the root of any of the bottom subtrees, if we know

- the size of the bottom subtree,
- the size of the corresponding top subtree, and
- the position of the root of the corresponding top subtree.

Since a node can only be the root of one top and one bottom subtree during the recursion, the sizes of bottom and top subtrees need only be calculated for each depth d in the tree. This can easily be done during the actual layout of the tree in memory, and stored in two arrays $P_T[d]$ and $P_B[d]$. These arrays are calculated in $O(\log_2 N)$ time, and take up $O(\log_2 N)$ space.

We know that the root of the entire tree is at position 1, so we start by calculating the position $Pos[2]$ of the bottom subtrees rooted at depth 2. These bottom subtrees must necessarily have a corresponding top subtree rooted at depth 1, so the position of the root of this top subtree is 1. Thus, we have

$$Pos[2] = Pos[1] + P_T[2] + xP_B[2], \tag{5.1}$$

since the root of the top subtree is also the parent. The x indicates the bottom subtree whose root position we are calculating. For the left-most subtree x is 0. When we descend further down the tree we can no longer be sure that the root of the corresponding top subtree is the parent, i.e., is located at current depth minus 1. Therefore, while building the arrays $P_T[d]$ and $P_B[d]$ we also keep track of the depth $P_D[d]$ at which the top trees are rooted. This only affects the time and space bounds of laying out the tree by a constant factor. Equation (5.1) can be generalized to

$$Pos[d] = Pos[P_D[d]] + P_T[d] + xP_B[d]. \tag{5.2}$$

The remaining problem is now to index the correct bottom subtree, i.e. to find the value of x. In other words, we need to calculate the path followed in the top subtree from its root to the root of the bottom subtree of interest. Here observation 2 will come in handy: When a node is visited during the search in the height partitioned layout, we can keep track of the position i_v of the corresponding node in the breadth-first layout.

If i_v is the position of the node v in the breadth-first layout, and v is the root of the bottom tree of interest, and is located at depth d_v in the height partitioned layout, then the binary and-operation $i_v \& P_T[d_v]$ will index the correct bottom subtree to descend in the height partitioned layout. Equation (5.2) becomes:

$$Pos[d] = Pos[P_D[d]] + P_T[d] + (i \& P_T[d])P_B[d]. \tag{5.3}$$

Given the precomputed arrays B, T, and D, the algorithm Height-Partitioning-Search searches the array A for the value *key*. Array A represents a tree in the height partitioned layout.

```
Height-Partitioning-Search($A, key, P_B, P_T, P_D$)
 1  $d \leftarrow 1$
 2  $Pos[1] \leftarrow 1$
 3  $i \leftarrow 1$
 4  while $P_D[d] \neq$ NIL
 5      do if $A[Pos[d]] = key$
 6            then return $true$
 7         if $A[Pos[d]] < key$
 8            then $i \leftarrow 2i + 1$
 9            else $i \leftarrow 2i$
10         $d \leftarrow d + 1$
11         $Pos[d] = Pos[P_D[d]] + P_T[d] + (i \& P_T[d]) \cdot P_B[d]$
12  return $false$
```

In line 1 to 3 d and $Pos[d]$ are initialized to the depth and position of the root. The while-loop (line 4 to 11) is executed at most once for each depth in the tree. In line 5 and 6, if the current node contains the value to be found, then the algorithm terminates by returning true. In line 7 to 9 the position in the breadth-first layout of the next node to search is calculated, and in line 10 and 11 the depth and position in the height partitioned layout of the next node to search is calculated.

As an example of Height-Partitioning-Search, we investigate the tree in Figure 5.8. The following table shows the precomputed arrays of this tree.

d	$P_B[d]$	$P_T[d]$	$P_D[d]$
2	15	1	1
3	1	1	2
4	3	3	2
5	1	1	4

Searching the tree for the value 18 involves examining Figure 5.9a and b as well as Figure 5.8. By the following computational steps, we calculate the positions of the nodes on the path from root to the 18-node:

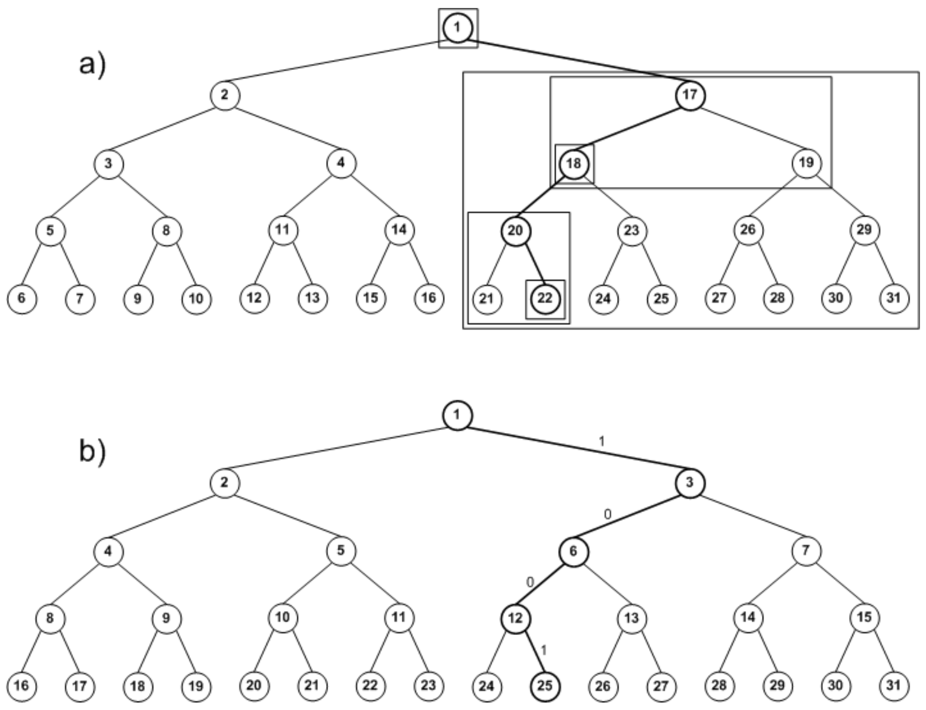


Figure 5.9: a) The root-to-node path visits memory position 1, 17, 18, 20, and 22 in the height partitioned layout, when searching for the value 18 in the tree on Figure 5.8. b) The corresponding positions of the nodes 1, 17, 18, 20, and 22 in the breadth-first layout. In HEIGHT-PARTITIONING-SEARCH the nodes are only conceptually laid out in the breadth-first layout. Actual storing of elements is only done according to the height partitioned layout.

$$
\begin{aligned}
&Pos[1] = 1\\
&Pos[2] = Pos[P_D[2]] + P_T[2] + (3 \& P_T[2]) \cdot P_B[2] = 17\\
&Pos[3] = Pos[P_D[3]] + P_T[3] + (6 \& P_T[3]) \cdot P_B[3] = 18\\
&Pos[4] = Pos[P_D[4]] + P_T[4] + (12 \& P_T[4]) \cdot P_B[4] = 20\\
&Pos[5] = Pos[P_D[5]] + P_T[5] + (25 \& P_T[5]) \cdot P_B[5] = 22
\end{aligned}
$$

Work Complexity

We have programmed the implicit height partitioning search both as a layout policy and in pure-C. As the HEIGHT-PARTITIONING-SEARCH algorithm indicates how the layout policy is implemented, we will here only show the pure-C implementation. In fact, Program 10 only shows the dominating while loop of this implementation, since only this part of the implementation is interesting when bounding the work complexity. The complete code for both implementations can be found in Appendix A.

```
 goto not_finished;
descend_right:
 current_bfs_index = current_bfs_index*2;
 current_bfs_index++;
 current_depth++;
 p = D + current_depth;
 x = *p;
 p = Pos + x;
 x = *p;
 p = T + current_depth;
 y = *p;
 z = current_bfs_index & y;
 p = B + current_depth;
 w = *p;
 z = z * w;
 w = x + y;
 w = w + z;
 p = Pos + current_depth;
 *p = w;
not_finished:
  if (current_bfs_index > size) goto return_false; /* hint: not taken */
node_contains:
 p = Pos+current_depth;
 x = *p;
 p = begin+x;
 x = *p;
 if (x == value) goto return_true;
 if (x < value) goto descend_right;
descend_left:
 current_bfs_index = current_bfs_index*2;
 current_depth++;
 p = D + current_depth;
 x = *p;
 p = Pos + x;
 x = *p;
 p = T + current_depth;
 y = *p;
 z = current_bfs_index & y;
 p = B + current_depth;
 w = *p;
 z = z * w;
 w = x + y;
 w = w + z;
 p = Pos + current_depth;
 *p = w;
 goto not_finished;
```

Program 10: The dominating while loop of the pure-C implementation of the implicit height partitioning search.

Theorem 11. *Searching among N elements laid out in memory according to the implicit height partitioned layout incurs at most $24\lfloor \log_2 N \rfloor + O(1)$ pure-C instructions, if using the generic search algorithm as defined in Program 2.*

Proof. In the worst case the search terminates in line 36 due to an unsuccessful search when `current_bfs_index` becomes larger than `size`. Descending down the left-most path in the tree incurs the least increment of `current_bfs_index`, and thereby the worst case situation. As the index in this case is doubled each iteration (beginning with a value of 1), the loop is executed $1 + \lfloor \log_2 N \rfloor$ times. As each iteration of the loop incurs 24 pure-C instructions the worst-case work complexity becomes $24\lfloor \log_2 N \rfloor + O(1)$. □

Branch Mispredictions

Theorem 12. *Searching in an array of N elements laid out in memory according to the implicit height partitioned layout incurs at most $\lfloor \log_2 N \rfloor + 2$ branch mispredictions, if using the generic search algorithm as defined in Program 2.*

Proof. Program 10 contains conditional branch instructions in line 36, 42, and 43. The branch instruction in line 36 has a hint, so it is mispredicted only once. The other two branch instructions have no hints, so they are mispredicted every second iteration of the loop, that is, $\frac{1+\lfloor \log_2 N \rfloor}{2}$ times each. Hence the implicit height partitioned layout causes $\lfloor \log_2 N \rfloor + 2$ branch mispredictions. □

Cache Complexity

The implicit height partitioned layout is different from the inorder and heap layouts in that it uses additional space to ease the navigational computations. This extra space usage makes the navigation influence the cache complexity of the implicit heap partitioned layout, which is not the case for the other layouts. Therefore, we have split the cache complexity analysis for this layout into three theorems. Theorem 13 states the number of cache misses incurred by the height partitioned tree itself, that is, not taking the navigational computations into account. Theorem 14 handles the navigational computations on their own and Theorem 15 states the total cache complexity of searching in the implicit height partitioned layout.

Theorem 13. *Searching in an array of N elements laid out in memory according to the implicit height partitioned layout incurs at most $Q(N) = 4\lfloor \log_B N \rfloor + 2$ cache misses, if the navigational computations are not considered.*

Proof. Suppose the tree is split recursively until every subtree has size at most B, i.e., until every subtree contains at most B and at least $\sqrt{B}$ nodes. Then, at this final level of recursion, each subtree is stored in an interval of memory of size at most B, and since the block alignment may not fit, each subtree occupies at most 2 blocks. All subtrees now have a height of at least $(\log_2 B)/2$ except the topmost top subtree which may be smaller. When the search traverses down the path from the root to a leaf a sequence of $2(\log_2 N)/(\log_2 B) = 2\log_B N$ recursive subtrees is visited in addition to the topmost subtree. Since visiting a subtree may incur up to 2 cache misses, the cost becomes $4\lfloor\log_B N\rfloor + 2$ misses. □

Theorem 14. *The navigational computations of searching in the implicit height partitioned layout incurs at most* $Q(N) = 4\left(\frac{\log_2 N}{B} + 1\right)$ *cache misses, provided* $M \geq 5B$.

Proof. During the search the navigational computations are supported by the arrays P_T, P_B, and P_D, each of size $\lfloor\log_2 N\rfloor$, and the array Pos of size $\lceil\log_2 N\rceil$. The computations at each depth $d > 1$ are given by Equation (5.3). According to this equation, a descend from the root to a leaf causes a scan of each of the four arrays — plus an additional lookup of $Pos[P_D[d]]$ at each depth. Ignoring floors and ceilings, the four scans incur $4\left(\frac{\log_2 N}{B} + 1\right)$ cache misses, provided that the cache can hold 4 blocks at a time, i.e., $M \geq 4$. The $\lceil\log_2 N\rceil$ lookups of $Pos[P_D[d]]$ incur no extra cache misses. Because $P_D[d] < d$ for all d (cf. the example on page 68), $Pos[P_D[d]]$ will always refer to an entry in Pos that have been brought into the cache earlier. If the cache can contain this extra block, i.e., $M \geq 5$, then the optimal replacement strategy of the ideal-cache model guarantees that the block has not been thrown out in the meantime. □

Theorem 15. *Searching in an array of N elements laid out in memory according to the implicit height partitioned layout incurs at most* $Q(N) = 4\lfloor\log_B N\rfloor + \frac{4\log_2 N}{B} + 6$ *cache misses, provided* $M \geq 6B$.

Proof. Follows directly from Theorem 13 and 14. □

5.7.2 Explicit Navigation

By use of explicit navigation we can omit the complex navigational computations and use pointers instead. Assuming 4-byte pointers and 4-byte elements the explicit layout is three times as space consuming as the implicit one, since each node has to contain pointers to its two children.

Work Complexity

Program 11 depicts the pure-C implementation using explicit navigation. The code for the corresponding layout policy is in Appendix A.

```
template <typename RandomIterator, typename T>
bool explicit_hp_search(RandomIterator begin,
                        RandomIterator beyond,
                        const T& value) {
 initialize:
  typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
  RandomIterator current_element = begin;
  T x;
  diff_type size = beyond - begin, current_bfs_index = 1;

  goto not_finished;
 descend_right:
  current_element = current_element->left_child;
  current_bfs_index = current_bfs_index*2;
  current_bfs_index++;
 not_finished:
  if (current_bfs_index > size) goto return_false; /* hint: not taken */
 node_contains:
  x = current_element->e;
  if (x == value) goto return_true;
  if (x > value) goto descend_right;
 descend_left:
  current_element = current_element->right_child;
  current_bfs_index = current_bfs_index*2;
  goto not_finished;

 return_false:
  return false;
 return_true:
  return true;
}
```

Program 11: The pure-C implementation of the explicit height partitioning search.

Theorem 16. *Searching among N elements laid out in memory according to the explicit height partitioned layout incurs at most $7\lfloor\log_2 N\rfloor + O(1)$ pure-C instructions, if using the generic search algorithm as defined in Program 2.*

Proof. The proof is similar to that of Theorem 11. Each of the $\lfloor\log_2 N\rfloor$ iterations incurs 7 pure-C instructions — remember, that line 13 and 19 both correspond to 2 pure-C instructions. □

Branch Mispredictions

Theorem 17. *Searching in an array of N elements laid out in memory according to the explicit height partitioned layout incurs at most $\lfloor\log_2 N\rfloor + 2$ branch mispredictions, if using the generic search algorithm as defined in Program 2.*

Proof. Program 11 contains conditional branch instructions similar to those of Program 10. Hence the explicit height partitioned layout causes the stated number of branch mispredictions. □

Cache Complexity

Also the cache complexity of using the explicit height partitioned layout resembles that of the implicit variant — except for the fact that it does not use any precomputed information.

Theorem 18. *Assuming that an element and a pointer to an element takes up the same amount of memory space, searching in an array of N elements laid out in memory according to the explicit height partitioned layout requires at most $Q(N) = 4\lfloor\log_B 3N\rfloor + 2$ cache misses.*

Proof. The proof is similar to that of Theorem 13. Though, since an element takes up three times as much space than is the case for the implicit layout we get the factor 3 of the $\lfloor\log_B 3N\rfloor$ term. □

5.7.3 Constructing a Height Partitioned Tree

As mentioned earlier, Ohashi [34] presented an algorithm for constructing a tree in the height partitioned layout. Given a sorted sequence of elements Ohashi's algorithm can construct a height partitioned tree in $O(N\log_2\log_2 N)$ time. In this section we present a linear time algorithm that performs the same task.

Given a source array of N sorted elements and a target array, also of size N, in which the tree is to be build, the algorithm works by splitting the target array into a top subtree T and a number of bottom subtrees $B_1, \ldots, B_k$. The algorithm then recursively splits the top tree and the bottom trees until the base case of the recursion is reached. As such, this splitting of the target array is conceptual, since no elements are actually moved until the bottom of the recursion is reached. In fact, the splitting can be viewed as a recursive mapping of the elements from the source to the target array, where the calculations of the positions of the elements in the target array become recursively more and more precise. In Figure 5.10 the mapping is shown for the coarsest level of recursion.

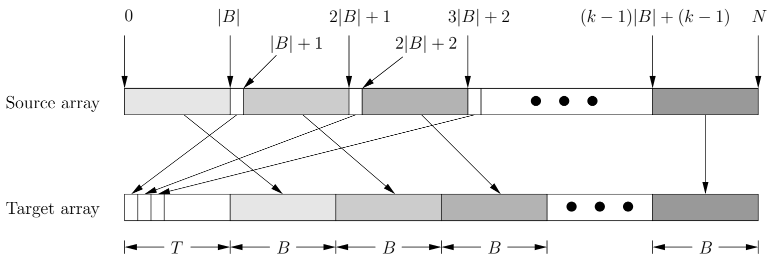


Figure 5.10: At the coarsest level of recursion we know that for every $|B|$ elements of the source array one element will eventually end up in the top tree in the target array. Furthermore, the elements within a bottom tree are placed continuously in both the source and the target array — though the final placement of the elements in the target array will be a permutation of the sequence in which they occurred in the source array. The mapping is not final, since only the first recursion of the conceptual mapping is shown. It only indicates in which area the elements will eventually end up.

For the bottom trees the mapping is easy. The elements of a bottom tree are placed continuously in the source array as well as in the target array, though the final placement of the elements in the target array will be a permutation of the sequence in which they occurred in the source array. This locality among the elements keeps the recursive mapping simple, since the segments of both the source and the target array will remain continuous.

For the top tree the mapping is more difficult. Let $|B|$ denote the size of a bottom tree. At the coarsest level of recursion we know that for every $|B|$ elements in the source array one element will belong to the top tree when the algorithm terminates; we just do not know yet at what position it ends up. Therefore, when we recurse the mapping of a top tree we have to keep track of $|B|$. Otherwise, we can not copy the elements to their correct position when the base case of the recursion is reached.

The algorithm has two base cases. At some point in the recursion the tree will either consist of a top tree and two bottom trees — each of height 1 — or of a single top tree containing just one element. The latter base case can occur if the height of the tree initially was not a power of 2 (for details, see the discussion of "odd-sized" subtrees on page 63). Program 12 is a

C++ program that builds a tree in the height partitioned layout. Initially, the program should be called with *step* $= 1$.

```
template<typename RandomIterator>
void build_hp_tree(RandomIterator begin_in,
                   RandomIterator begin_out,
                   int height,
                   int step) {
  int bottom_height = height==2?1:hyperfloor(height-1);
  int top_height    = height-bottom_height;
  int bottom_size   = (1<<(bottom_height))-1;
  int top_size      = (1<<(top_height))-1;

  if (top_height == 1 && bottom_height == 1) {
    begin_out[1] = begin_in[0];
    begin_out[0] = begin_in[1*step];
    begin_out[2] = begin_in[2*step];
    return;
  }
  if (top_height == 1) {
    begin_out[0] = begin_in[bottom_size*step];
  } else {
    build_hp_tree(begin_in+bottom_size*step,
                  begin_out,
                  top_height,
                  bottom_size*step+step);
  }
  for(int i = 0; i <= top_size; i++) {
    build_hp_tree(begin_in+(i*bottom_size+i)*step,
                   begin_out+top_size+i*bottom_size,
                   bottom_height,
                    step);
  }
}
```

Program 12: C++ program implementing the algorithm for building a tree in the height partitioned layout.

Theorem 19. *Assuming that N is $2^i - 1$ for some positive integer i, Program 12 builds a tree of N elements in the height partitioned layout in $O(N)$ time.*

Proof. The height h of the tree is given by $\log_2(N+1)$, which implies $N = 2^h - 1$. In each recursive step the input is divided into a single top tree of size $2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1$ and $2^{h-2^{\lceil \log_2 h/2 \rceil}}$ bottom trees each of size $2^{2^{\lceil \log_2 h/2 \rceil}} - 1$. We denote the work of subdividing the problem by $d2^{h-2^{\lceil \log_2 h/2 \rceil}}$. Hence, we have the recurrence

$$T(N) = T(2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1) + (2^{h-2^{\lceil \log_2 h/2 \rceil}})T(2^{2^{\lceil \log_2 h/2 \rceil}} - 1) + d2^{h-2^{\lceil \log_2 h/2 \rceil}}. \quad (5.4)$$

We want to prove that $T(N) \leq cN$ for some constant $c > 0$. We assume that this bound holds for $2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1$ and $2^{2^{\lceil \log_2 h/2 \rceil}} - 1$, that is,

$$T(2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1) \leq c\left(2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1\right)$$

and

$$T(2^{2^{\lceil \log_2 h/2 \rceil}} - 1) \leq c\left(2^{2^{\lceil \log_2 h/2 \rceil}} - 1\right).$$

Substituting these values into Recurrence (5.4) yields

$$\begin{aligned}
T(N) &\leq c(2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1) + 2^{h-2^{\lceil \log_2 h/2 \rceil}} c(2^{2^{\lceil \log_2 h/2 \rceil}} - 1) + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= c(2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1) + c2^h - c2^{h-2^{\lceil \log_2 h/2 \rceil}} + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= c2^h - c + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= c(N+1) - c + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= cN + d2^{h-2^{\lceil \log_2 h/2 \rceil}},
\end{aligned}$$

which does not imply $T(N) \leq cN$ for any choice of c. We therefore try with the stronger inductive hypothesis $T(N) \leq cN - b$, where b is some constant greater than or equal to 0. Again, assuming that this bound holds for $2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1$ and $2^{2^{\lceil \log_2 h/2 \rceil}} - 1$, and substituting these values into Recurrence (5.4) yields

$$\begin{aligned}
T(N) &\leq c(2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1) - b + 2^{h-2^{\lceil \log_2 h/2 \rceil}} \left(c(2^{2^{\lceil \log_2 h/2 \rceil}} - 1) - b\right) \\
&\qquad + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= c2^{h-2^{\lceil \log_2 h/2 \rceil}} - c - b + c2^h - c2^{h-2^{\lceil \log_2 h/2 \rceil}} - b2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&\qquad + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= c2^h - c - b - b2^{h-2^{\lceil \log_2 h/2 \rceil}} + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= c(N+1) - c - b - b2^{h-2^{\lceil \log_2 h/2 \rceil}} + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&= cN - b - b2^{h-2^{\lceil \log_2 h/2 \rceil}} + d2^{h-2^{\lceil \log_2 h/2 \rceil}} \\
&\leq cN - b,
\end{aligned}$$

which holds when $b \geq d$. As this bound also holds for the base cases $T(1)$ and $T(3)$ if we choose $c > 1$ and $b = 0$ we are done. □

Theorem 20. *Assuming that N is $2^i - 1$ for some positive integer i, building a tree of N elements in the height partitioned layout with Program 12 incurs at most $Q(N) = 2\sqrt{N+1} + \frac{\sqrt{N+1}}{B} + 2\frac{N+1}{B}$ cache misses.*

Proof. We prove this bound by looking at the cost of constructing the top tree and the bottom trees respectively. In the following, we assume that h is the height of the tree defined as $\log_2(N+1)$.

As depicted on Figure 5.10, the elements that belong to the top tree are scattered throughout the source array. In the worst case none of these elements belong to the same cache block, so accessing each of these $2^{h-2^{\lceil \log_2 h/2 \rceil}} - 1$ elements in the source array costs at most 2 misses per element (due to alignment). Writing these elements to the target array costs at most $\frac{2^{h-2^{\lceil \log_2 h/2 \rceil}}}{B}$ misses, so the total cost of constructing the top tree is

$$2 \cdot 2^{h-2^{\lceil \log_2 h/2 \rceil}} + \frac{2^{h-2^{\lceil \log_2 h/2 \rceil}}}{B}.$$

The elements of the bottom trees are placed continuously in both source and the target array, so copying a single bottom tree incurs $2 \cdot 2^{2^{\lceil \log_2 h/2 \rceil}}/B$ misses. For all bottom trees this cost adds up to

$$2 \cdot 2^{h-2^{\lceil \log_2 h/2 \rceil}} \cdot \frac{2^{2^{\lceil \log_2 h/2 \rceil}}}{B}.$$

The total number of cache misses of laying out the tree becomes

$$\begin{aligned} Q(N) &= 2 \cdot 2^{h-2^{\lceil \log_2 h/2 \rceil}} + \frac{2^{h-2^{\lceil \log_2 h/2 \rceil}}}{B} + 2 \cdot 2^{h-2^{\lceil \log_2 h/2 \rceil}} \cdot \frac{2^{2^{\lceil \log_2 h/2 \rceil}}}{B} \\ &= 2 \cdot 2^{h-2^{\lceil \log_2 h/2 \rceil}} + \frac{2^{h-2^{\lceil \log_2 h/2 \rceil}}}{B} + 2 \cdot \frac{2^h}{B} \\ &\leq 2\sqrt{N+1} + \frac{\sqrt{N+1}}{B} + 2\frac{N+1}{B}. \end{aligned}$$

□

The assumptions of Theorem 19 and 20, that N has to be in the form $2^i - 1$ (that is, the tree has to be complete), can easily be relaxed by building a height partitioned tree a little bigger than actually needed, i.e., of size $2^{\lceil log_2 N \rceil} - 1$, and allowing empty nodes and leaves in the tree. In the worst case a tree containing N elements will be of size $2N - 1$, which implies a worst-case space overhead of size $N - 1$.

5.8 Expected Performance

The work complexities of the heap layouts are difficult to compare to those of the other three layouts, since the use of the lower bound algorithm makes the constant B influence the constant factor in the most significant terms of the pure-C instruction count for those two layouts. On our benchmarking machines the cache line length is 64 bytes, so for 4-byte elements B is 16. For this value of B the work and cache complexities of all five layouts are shown in Figure 5.11. The branch misprediction counts are shown in Figure 5.12. We have rewritten the complexities and counts into base 2 logarithm to make them easier to compare. This way of presenting the characteristics of the layouts gives us a better idea of what we can expect from the benchmarks in Chapter 7. The change of logarithm base reveals a couple interesting aspects of the five layouts that we will describe in the following sections.

5.8.1 Knowing the Memory System

For the implicit heap layout, knowing the characteristics of the memory system seems to pay off. The implicit heap layout has a work complexity similar to those of the inorder layout and the explicit layouts, and its cache complexity is the best among the five. The branch misprediction counts of both heap layouts are a little higher than for the other layouts. This difference is due to the lower bound calculations used within the nodes of these layouts.

5.8.2 Alignment of Data

The inorder layout is superior to the implicit height partitioned layout in both work and cache complexity. It is no surprise that the implicit height partitioned layout exhibits worse work complexity, but it is disappointing that its cache complexity is inferior to that of the inorder layout, since the whole idea of the height partitioned layout was to increase cache performance. The bad cache complexity of the implicit height partitioned layout is partly due to the extra space used for the navigational computations. Nevertheless, even though we ignore the contribution of the navigation (cf. Theorem 13) the cache complexity is still worse, since $\lceil \log_2 N \rceil - 6$ is smaller than $4\lfloor \log_{16} N \rfloor + 2 \approx \lfloor \log_2 N \rfloor + 2$, for all $N > 0$.

Figure 5.11 shows the worst-case cache complexity bounds, so the worst possible data alignment is assumed. In practice, though, data alignment this is rarely the case. In the implicit height partitioned layout we keep on splitting the tree until the size of the subtrees is smaller than B (see the

Layout	Work complexity	Cache complexity
Inorder	$10 \log_2 N + O(1)$	$\lceil \log_2 N \rceil - 6$
Imp. Heap	$12 \log_2 N + O(1)$	$\lceil \frac{\log_2 N}{\log_2 17} \rceil \leq \lceil \frac{\log_2 N}{4} \rceil$
Exp. Heap	$11 \log_2 N + O(1)$	$\lceil \frac{\log_2 N}{3} \rceil$
Imp. Height Part.	$24 \log_2 N + O(1)$	$4\lfloor \log_{16} N \rfloor + \frac{4 \log_2 N}{16} + 6$
		$\leq \frac{5 \log_2 N}{4} + 6$
Exp. Height Part.	$9 \log_2 N + O(1)$	$4\lfloor \log_{16} 3N \rfloor + 2$
		$\leq \lfloor \log_2 3N \rfloor + 2$

Figure 5.11: The work and cache complexities of the five layouts. The logarithms are rewritten into base 2 to make the comparison easier.

Layout	Branch mispredictions
Inorder	$\lfloor \log_2 N \rfloor + 2$
Imp. Heap	$\frac{1}{2}\left(\lfloor \log_{(B+1)} N \rfloor + 1\right)(\log_2 B + 4) + 1$
	$= 4\lfloor \log_{17} N \rfloor + 5 \approx \lfloor \log_2 N \rfloor + 5$
Exp. Heap	$\frac{1}{2}\left(\lfloor \log_{(B/2)} N \rfloor + 1\right)(\log_2 (B/2 - 1) + 4) + 1$
	$\leq 4\lfloor \log_8 N \rfloor + 5 = \frac{4}{3}\lfloor \log_2 N \rfloor + 5$
Imp. Height Part.	$\lfloor \log_2 N \rfloor + 2$
Exp. Height Part.	$\lfloor \log_2 N \rfloor + 2$

Figure 5.12: The branch misprediction counts of the five layouts. The logarithms are rewritten into base 2 to make the comparison easier.

Cache-line length	B	subtree size	% badly aligned subtrees
32 bytes	8	$2^{2^1} - 1 = 3$	$2/8 = 25$
64 bytes	16	$2^{2^2} - 1 = 15$	$14/16 = 87.5$
128 bytes	32	$2^{2^2} - 1 = 15$	$14/32 = 43.75$
256 bytes	64	$2^{2^2} - 1 = 15$	$14/128 = 10.94$

Figure 5.13: For 4-byte elements $B = 16$ results in the largest percentage of badly aligned subtrees.

cache complexity analysis on page 71). Depending on the cache-line length, all subtrees (except for the top-most top subtree) will then have a size larger than $\sqrt{B}$ and smaller than B. The smaller the subtree, the smaller risk of it being badly aligned and therefore occupying 2 cache lines.

Unfortunately, when $B = 16$, as is the case on the benchmark computers, a subtree will be smaller than B when it reaches size 15. The consequence of this is that the majority of the subtrees will be aligned badly. In fact, on average $14/16 = 87.5\%$ of the subtrees will have elements in two cache lines. If we had $B = 32$, then the situation would have been better. The subtrees would still end up having size 15, but on average only $14/32 = 43.75\%$ of the subtrees would be aligned badly. Figure 5.13 shows the average percentage of badly aligned subtrees for the height partitioned layout for various cache-line lengths. Recall, that splitting the height partitioned tree results in subtrees of size $2^{2^i} - 1$. Since $2^{2^2} = 16$ the 64-byte cache line is the worst possible for 4-byte elements.

In contrast to the implicit height partitioned layout the inorder layout is almost completely unaffected by data alignment. No matter how data is aligned the first $\lceil \log_2 N \rceil - \lfloor \log_2 B \rfloor$ levels in the tree will access one cache line each. Only the final $\log_2 B$ levels may incur a single additional miss due to bad alignment.

5.8.3 Implicit vs. Explicit Navigation

As one might expect, the use of explicit navigation decreases the pure-C instruction count but makes the cache complexity worse. This is apparent for both the heap and the height partitioned layouts. It is hard to predict whether the lower instruction count can pay for the inferior cache complexity, since the situation may change when the dataset sizes exceeds the different memory layer sizes. For example, the lower instruction count may pay off as long as the dataset fits within the level 2 cache, because level 1 cache misses are quite inexpensive, while the better cache complexity may start paying off when the dataset exceeds the size of the level 2 cache, as misses at that level cost more CPU cycles.

5.9 Summary

In this chapter we have investigated five different static search tree layouts. We have presented a generic search algorithm and the notion of layout policies that together make it possible to separate the memory layout of data from the overall structure of the search algorithm. By use of this strategy we have succeeded in conducting a comparative analysis of all five layouts with respect to the constant factors of work complexity and cache complexity. In order to derive the constant factors of the work complexities we have translated all search algorithms into full running pure-C programs.

Furthermore, we have presented a new algorithm for constructing a tree in the cache-efficient height partitioned layout. The algorithm runs in linear time and incurs at most $Q(N) = 2\sqrt{N+1} + \frac{\sqrt{N+1}}{B} + 2\frac{N+1}{B}$ cache misses.

CHAPTER 6

Sorting

"The most damaging phrase in the language is: It's always been done that way."
— Grace Hopper

Given an array of N elements the *sorting problem* consists of reordering the elements such that they appear in nondecreasing order. Many different approaches can successfully be applied to solve the sorting problem. The sorting algorithm insertionsort, works in a way similar to that of a card player arranging his hand, picking up one card at a time and inserting it at the appropriate position relative to the previously-arranged cards. Other sorting algorithms, such as bubblesort and quicksort, work by exchanging elements, that is, whichever two elements that are found to be out of order are interchanged. The process continues until no more interchanges are necessary.

In this chapter our main goal is to analyze the cache-oblivious *lazy funnelsort* algorithm of Brodal and Fagerberg [12]. This algorithm can be considered a variant of mergesort, and is a modification of the original *funnelsort* by Frigo et al. [19]. The lazy funnelsort is basically the same algorithm as the original, but both its description and analysis are simpler.

In order to determine the efficiency of funnelsort[1] we compare it theoretically to 4-way mergesort, which is a more traditional mergesort variant. Our investigation covers constant-factors analysis of cache-efficiency in the ideal-cache model as well as work complexity in the pure-C model. We analyze 4-way mergesort in Section 6.1.

Central to funnelsort is a static data structure called a k-funnel. It is important to understand how this data structure operates in order to

[1]In the following we will refer to lazy funnelsort simply as funnelsort.

understand the workings of funnelsort. Therefore, we introduce the k-funnel in Section 6.2, prior to describing the funnelsort algorithm in Section 6.3. In Section 6.4 we analyze both the k-funnel and the funnelsort algorithm.

In Section 6.5 we compare the two sorting algorithms theoretically and comment on what can be expected from the benchmarks in Chapter 7.

6.1 The 4-way Mergesort

Mergesort is a term used for a family of sorting algorithms that all operate in a similar way. The common approach used by mergesort algorithms is to consider an initially unsorted array of N elements as N sorted subarrays containing each one element, and then merge these subarrays recursively until only a single sorted array remains. The 2-way divide-and-conquer mergesort is considered the basic mergesort algorithm and operates as follows:

Divide: Divide the N-element array to be sorted into two subarrays of $N/2$ elements each.

Conquer: Sort the two subarrays recursively using 2-way mergesort.

Combine: Merge the two, now sorted, subarrays into one sorted array.

For its simplicity and efficiency the 2-way mergesort is a widely used sorting algorithm. It is optimal in terms of asymptotic work complexity [17], and according to Demaine [18] its cache complexity in the ideal-cache model is $O(N/B \log_2 N/B)$, which is close to the optimal $O(N/B \log_{M/B} N/B)$ complexity bound for comparison-based sorting [3].

Other mergesort variants exist. Katajainen & Träff [23] analyzed 2-, 3-, and 4-way mergesort algorithms in the pure-C cost model, and according to their empirical measurements the 4-way mergesort algorithm is the most efficient. These results are supported by Mortensen [31] who finds the same 4-way mergesort implementation the most efficient among a number of mergesort variants.

The 4-way mergesort works in a way similar to the 2-way mergesort, the difference being that it divides the array into 4 subarrays and therefore merges 4 subarrays at a time instead of 2. This similarity makes it reasonable to expect that it too exhibits good cache efficiency. Along with the fact that the algorithm works well in practice, we are convinced that it is well suited as a competitor to funnelsort.

The following 4-way mergesort algorithm sorts the elements in the subarray $A[p..q)$. If a subarray contains less than 4 elements it is sorted using insertionsort. Otherwise, the divide step partitions $A[p..q)$ into 4 subarrays of approximately equal size, recursively sorts these subarrays, and merges them.

```
4-WAY-MERGESORT($A, p, q$)
1  $len \leftarrow q - p$
2  if $len > 3$
3     then $fourth = len/4$
4           4-WAY-MERGESORT($A, p, p + fourth$)
5           4-WAY-MERGESORT($A, p + fourth, p + 2 \cdot fourth$)
6           4-WAY-MERGESORT($A, p + 2 \cdot fourth, p + 3 \cdot fourth$)
7           4-WAY-MERGESORT($A, p + 3 \cdot fourth, q$)
8           4-WAY-MERGE($p, p + fourth, p + 2 \cdot fourth, p + 3 \cdot fourth$)
9     else  INSERTIONSORT($A, p, q$)
```

The procedure 4-WAY-MERGE uses an additional array of size N to which the sorted subarrays are merged. To avoid copying the elements back to $A[p..q)$ after a merge step, an implementation of 4-WAY-MERGESORT makes sure to merge the elements forth and back between $A[p..q)$ and the additional array.

6.1.1 Work Complexity and Branch Mispredictions

Both Mortensen and Katajainen & Träff deduce the number of pure-C instructions and branch mispredictions incurred by 4-WAY-MERGESORT. For an input of size N the algorithm executes $3.25N \log_2 N + O(N)$ pure-C instructions and causes $0.53N \log_2 N + O(N)$ branch mispredictions in the worst-case[2].

6.1.2 Cache Complexity

Mortensen also analyzed the cache complexity of the algorithm, but since he did not assume the ideal-cache model we will redo the analysis here. Theorem 21 states the cache complexity of the 4-WAY-MERGESORT.

Theorem 21. *Sorting N elements with the 4-way mergesort algorithm incurs $Q(N) < 2\frac{N}{B} \log_4 \frac{N}{M} + 6\frac{N}{B} - 4 \log_4 \frac{N}{B} + 4$ cache misses, provided that $M \geq 8B$.*

Proof. In the proof we use use $\log_b a$ as a shorthand for $\max\{\frac{\ln b}{\ln a}\}$, which is a common assumption when analyzing recursive structures.

The call to 4-WAY-MERGE merges the 4 subarrays $A[p..p + fourth)$, $A[p + fourth..p + 2 \cdot fourth)$, $A[p + 2 \cdot fourth..p + 3 \cdot fourth)$ and $A[p + 3 \cdot fourth..q)$ to a temporary array also of length N. In the next call the procedure merges the elements back to $A[p..q)$. In the following we only consider what happens when we merge from $A[p..q)$ to the temporary array, but the description also holds the other way around.

[2]For a detailed analysis of the carefully written pure-C programs that adhere to these bounds we refer to [23, 31].

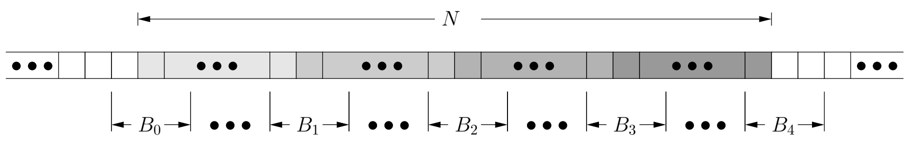


Figure 6.1: In case of bad data alignment the first blocks of the 3 last subarrays are the same as the last blocks of the first 3 subarrays, namely block B_1, B_2, and B_3. By keeping these 3 blocks in the cache during the merge we can avoid that they are read into the cache twice.

We assume that $A[p..q)$ is laid out in memory in continuous locations and that the cache can contain at least 8 blocks simultaneously. During the merge the cache contains 1 block of each of the 4 subarrays that are being merged at that time and one block of the target array. The contents of these five cache lines change as the merging progresses.

In the remaining 3 cache lines, at any time during the merge, the first block of the last 3 subarrays is contained.

In case of bad data alignment the first blocks of the last 3 subarrays are the same blocks as the last blocks of the first 3 subarrays. Therefore, by keeping the first blocks of the last 3 subarrays in the cache after the merging of their elements, we can avoid that these blocks are read once again when the merge reaches the ends of the subarrays. The situation is shown in Figure 6.1.

For $N \geq M$, merging the 4 subarrays corresponds to scanning 2 sequences of N elements, which incurs $2\lceil \frac{N}{B} \rceil + 2 < 2\frac{N}{B} + 4$ cache misses in the worst case. For $N < M$ the cache can contain the 4 subarrays entirely, but we still need to read and write the elements, which incurs $2\lceil \frac{N}{B} \rceil + 2 < 2\frac{N}{B} + 4$ cache misses. The total number of cache misses $Q(N)$ incurred by the 4-WAY-MERGESORT algorithm is therefore given by the recurrence

$$Q(N) < \begin{cases} 2\frac{N}{B} + 4 & \text{if } N < M, \\ 4Q(\frac{N}{4}) + 2\frac{N}{B} + 4 & \text{otherwise.} \end{cases}$$

For the recursive case we assume that the bound $Q(N) < 2\frac{N}{B}\log_4 \frac{N}{M} + 6\frac{N}{B} - 4\log_4 \frac{N}{B} + 4$ holds for $\frac{N}{4}$. Substituting into the recurrence yields

$$\begin{aligned} Q(N) &< 4\left(2\frac{N/4}{B}\log_4 \frac{N/4}{M} + 6\frac{N/4}{B} - 4\log_4 \frac{N/4}{B} + 4\right) + 2\frac{N}{B} + 4 \\ &= 2\frac{N}{B}(\log_4 N - 1 - \log_4 M) + 6\frac{N}{B} \\ &\qquad -16(\log_4 N - 1 - \log_4 B) + 2\frac{N}{B} + 20 \\ &= 2\frac{N}{B}\log_4 \frac{N}{M} - 2\frac{N}{B} - 16\log_4 \frac{N}{B} + 8\frac{N}{B} + 20 \end{aligned}$$

$$
\begin{aligned}
&= 2\frac{N}{B}\log_4\frac{N}{M} + 6\frac{N}{B} - 16\log_4\frac{N}{B} + 4 \\
&< 2\frac{N}{B}\log_4\frac{N}{M} + 6\frac{N}{B} - 4\log_4\frac{N}{B} + 4.
\end{aligned}
$$

We split the base case $N < M$ into five separate cases:

Case I: When $4B \leq N < M$ we have

$$
\begin{aligned}
& 2\frac{N}{B}\log_4\frac{N}{M} + 6\frac{N}{B} - 4\log_4\frac{N}{B} + 4 \qquad (6.1)\\
> \;& 2\frac{N}{B} + 6\frac{N}{B} \\
> \;& 2\frac{N}{B} + 4,
\end{aligned}
$$

since $\log_4\frac{N}{M} = 1$ and $4 - 4\log_4\frac{N}{B} \geq 0$.

Case II: When $3B \leq N < 4B$, the value of Expression (6.1) is at least 24. This is larger than the value of $2\frac{N}{B} + 4$, which is less than 12.

Case III: When $2B \leq N < 3B$ the value of Expression (6.1) is at least 16. This is larger than the value of $2\frac{N}{B} + 4$, which is less than 10.

Case IV: When $B \leq N < 2B$ the value of Expression (6.1) is at least 8. This is larger than the value of $2\frac{N}{B} + 4$, which is less than 8.

Case V: When $N < B$ the value of Expression (6.1) is at least 8. This is larger than the value of $2\frac{N}{B} + 4$, which is less than 6.

This concludes the proof of Theorem 21. □

6.2 The k-funnel Data Structure

The k-funnel data structure offers a way of merging k sorted sequences each containing k^{d-1} elements, where $d \geq 2$, into a single sorted sequence of size k^d. The k-funnel merges these k^d elements cache efficiently and cache-obliviously by ensuring that the merging process at all times works on data that is stored in memory as locally as possible.

A k-funnel is a complete binary tree with k leaves where these leaves are connected to the k sorted input lists. Therefore, k is assumed to be of the form 2^i for some positive integer i. Each internal node in the tree is a binary merger and each edge between two internal nodes contains a buffer. Each of these buffers acts as an output buffer for the binary merger in the lower node and as one of the two input buffers for the binary merger in the upper node. The k^d elements that are output from merger tree have been merged on their way up the tree — from the sorted input streams at the leaves to the output buffer of the binary merger at the root.

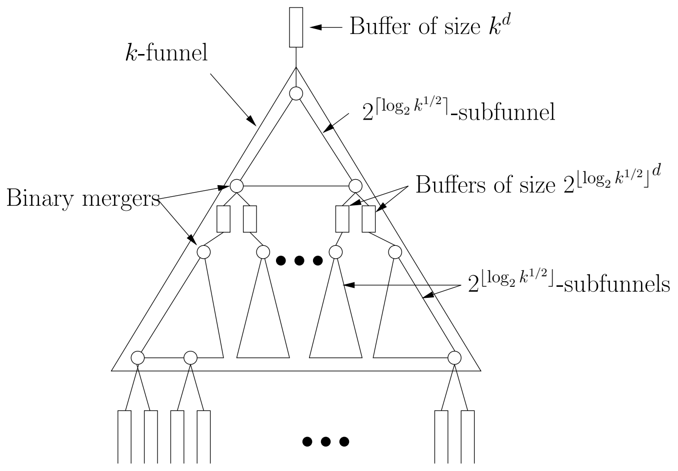

Buffer of size k^d
k-funnel
$2^{\lceil \log_2 k^{1/2} \rceil}$-subfunnel
Binary mergers
Buffers of size $2^{\lfloor \log_2 k^{1/2} \rfloor d}$
$2^{\lfloor \log_2 k^{1/2} \rfloor}$-subfunnels

Figure 6.2: A k-funnel is a complete binary tree with binary mergers in every node and buffers on the edges connecting the output of a merger in a lower node to the input of a merger in an upper node.

The k-funnel is a recursive data structure, so the size of the internal buffers are defined recursively in terms of subfunnels of smaller size. By the description of Brodal and Fagerberg the k-funnel is conceptually split between the nodes at depth $\lceil i/2 \rceil$ and the nodes at depth $\lceil i/2 \rceil + 1$ into a top recursive $2^{\lceil \log_2 k^{1/2} \rceil}$-subfunnel and $2^{\lceil \log_2 k^{1/2} \rceil}$ bottom recursive $2^{\lfloor \log_2 k^{1/2} \rfloor}$-subfunnels [3] . The buffers that connect the bottom subfunnels to the top subfunnel need to hold all the elements that the bottom subfunnels output, so they are each of size $2^{\lfloor \log_2 k^{1/2} \rfloor^d}$. Within the subfunnels the sizes of the buffers are recursively defined.

The way of splitting the k-funnel resembles the way we split the height partitioned tree in Chapter 5. But where we split the height partitioned tree in such a way that the top subtree is never larger than the bottom subtrees, it is the other way around for the top and bottom subfunnels: An uneven split results in a top subfunnel that is larger than the bottom subfunnels. Figure 6.2 shows the concept of a k-funnel for this way of splitting.

Also the way in which the buffers of the k-funnel are laid out in memory in continuous locations resembles the layout of the height partitioned search tree. This means, that for a k-funnel, the layout of the top subfunnel is followed by the layout of the bottom subfunnels. The output buffer of a bottom subfunnel is placed just before the bottom subfunnel itself. Figure 6.3 shows the buffer sizes and illustrates the memory layout of a 16-funnel for $d = 3$, that is, a k-funnel with 16 input buffers that outputs $16^3 = 4096$ elements during an invocation.

[3]Actually, Brodal & Fagerberg round off these values to $\sqrt{k}$, so the k-funnel consists of $\sqrt{k} + 1$ $\sqrt{k}$-subfunnels.

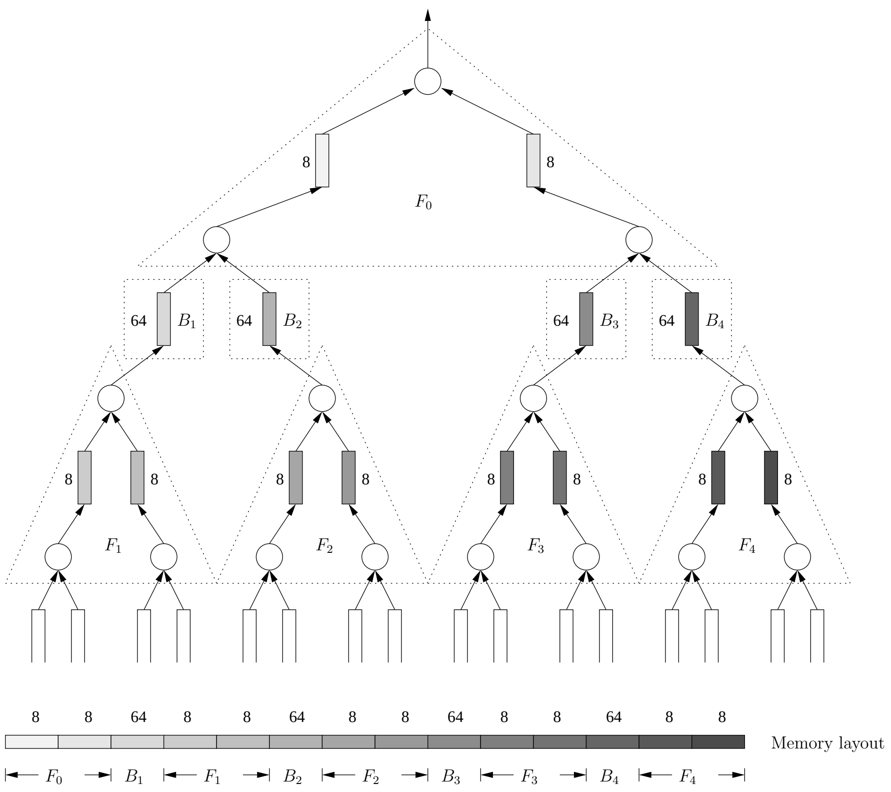


Figure 6.3: The layout of a 16-funnel for $d = 3$. The buffers are recursively laid out in memory in continuous locations, first the top subfunnel, then the bottom subfunnels. The output buffer of a bottom subfunnel is placed just before the bottom subfunnel itself. The 16-funnel is first split between depth 2 and 3, whereupon the resulting top subfunnel and bottom subfunnels are split between depth 1 and 2, and depth 3 and 4 respectively.

We delay the analysis of the k-funnel to after the description of the funnelsort algorithm. For now, it suffices to think of a k-funnel as a buffered merge tree like the one just described, and that it can merge k^d elements incurring only $O(\frac{k^d}{B}\log_M k^d + k)$ cache misses. We prove this bound in Section 6.4.2.

6.2.1 Creating a k-funnel

The following procedure Allocate-Funnel creates a k-funnel by connecting binary-merger objects and buffer objects into the desired balanced tree structure. The nodes are expected to be binary-merger objects holding pointers to two input buffers, an output buffer, and two sources, that are the binary mergers just below the node.

The procedure first creates an array S of k empty binary-merger objects (nil-objects). These empty mergers will eventually end up as sources to the binary mergers just above the buffers at the leaf level of the entire k-funnel. Allocate-Funnel then calls the recursive Allocate-Funnel-Rec procedure that builds up the funnel. In the base case, when $k = 2$, this procedure returns a binary-merger object with pointers to its two continuous input buffers $I[0]$ and $I[1]$, a pointer to its output buffer O, and pointers to its source nodes $S[0]$ and $S[2]$. In the recursive case, Allocate-Funnel-Rec first allocates space for the k middle buffers, each of size $k^{(d+1)/2}$, and then allocates the subfunnels that uses these middle buffers as output buffers. Finally, the top subfunnel is allocated.

```
Allocate-Funnel($I, k$)
1  for $i \leftarrow 0$ to $k-1$
2      do $S[i] \leftarrow$ Create-Binary-Merger(nil)
3  return Allocate-Funnel-Rec($O, k, I, S$)
```

```
Allocate-Funnel-Rec($O, k, I, S$)
 1  if $k = 2$
 2     then return Create-Binary-Merger($I[0], I[1], O, S[0], S[1]$)
 3     else $h \leftarrow \lfloor \log_2 k \rfloor$
 4          $k_b \leftarrow 2^{h/2}$
 5          $k_t \leftarrow h - k_b$
 6          for $i \leftarrow 0$ to $k-1$
 7              do $M[i] \leftarrow$ Allocate-Buffer($k^{(d+1)/2}$)
 8          for $i \leftarrow 0$ to $k-1$
 9              do $i_b \leftarrow i \cdot k_b$
10                 $i_e \leftarrow$ Length($I$)
11                 $B[i] \leftarrow$ Allocate-Funnel-Rec($M[i], k_b, I[i_b..i_e), S[i_b..i_e)$)
12          return Allocate-Funnel-Rec($O, k_t, M, B$)
```

6.2.2 Invoking a k-funnel

Within each node in the k-funnel the following merge process is executed in order to fill the output buffer of a merger. The node is a binary-merger object, where b_{left} and b_{right} are the two input buffers, b_{output} is the output buffer, and s_{left} and s_{right} are the sources:

```
FILL($node$)
1  while $node.b_{output}$ is not full
2     do if $node.b_{left}$ is empty
3           then FILL($node.s_{left}$)
4        if $node.b_{right}$ is empty
5           then FILL($node.s_{right}$)
6        if FIRST($node.b_{right}$) < FIRST($node.b_{left}$)
7           then move FIRST($node.b_{right}$) to $node.b_{output}$
8           else move FIRST($node.b_{left}$) to $node.b_{output}$
```

A single invocation of a k-funnel, that outputs k^d sorted elements, is simply a call to FILL on the root of the entire k-funnel. It follows from the FILL procedure that the buffers are filled completely before they are emptied, and that buffers are emptied completely before they are refilled.

6.3 The Cache-Oblivious Funnelsort

The $O(\frac{N}{B}\log_4\frac{N}{M})$ cache complexity of 4-WAY-MERGESORT is a bit closer to the optimal $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ bound on comparison-based sorting than the cache complexity of standard 2-way mergesort. The cache-oblivious *funnelsort* algorithm matches this optimal bound.

As a k-funnel merges k^d elements very cheaply — incurring $O(\frac{k^d}{B}\log_M k^d + k)$ cache misses — we could be tempted to use an N-funnel to sort N elements. We could simply have N sorted lists of 1 element each and use an N-funnel to sort them. This would incur only $O(\frac{N}{B}\log_M N + N^{1/d})$ cache misses. However, the technique used by the funnel that makes it work on data in a local manner cannot guarantee this complexity bound when a k-funnel outputs less than k^d elements, so merging N elements with an N-funnel is not a good idea. Furthermore, a k-funnel uses approximately $ck^{(1+d)/2}$ space [12], where $c \geq 1$, and since d has to be larger than or equal to 2, it would not result in an algorithm of linear space complexity.

Instead of using a single N-funnel to sort N elements, the FUNNELSORT algorithm uses funnels recursively in a multiway merge. First it splits the input array $A[p..q)$ into $k = (q-p)^{1/d}$ contiguous segments each of size $m = (q-p)^{1-1/d}$. Then it recursively sorts each segment, whereupon it merges the now sorted segments using a k-funnel. The smallest possible k-funnel is a 2-funnel that outputs 2^d elements, so the base case of the algorithm is reached when the subarray to sort contains less than 2^d elements. For these small subproblems any other sorting algorithm can be applied (e.g., the 4-way mergesort algorithm, or the introsort algorithm of Musser [33] available at SGI STL):

```
FUNNELSORT($A, p, q$)
 1  $len \leftarrow q - p$
 2  if $len < 2^d$
 3     then ANY-OTHER-SORT($A, p, q$)
 4     else  $k \leftarrow len^{1/d}$
 5           $m \leftarrow len^{1-1/d}$
 6           $i \leftarrow 0$
 7           $start \leftarrow p$
 8           while $start < q$
 9               do FUNNELSORT($A, start, start + m$)
10                  $I[i] \leftarrow$ CREATE-BUFFER($start, start + m$)
11                  $start \leftarrow start + m$
12                  $i \leftarrow i + 1$
13           $root \leftarrow$ ALLOCATE-FUNNEL($I, k$)
14           FILL($root$)
```

The procedure CREATE-BUFFER returns an object that allows us to treat the interval from *start* to $start + m$ of A as one of the sorted input buffers that are about to be merged by a funnel in line 14. Thus, the result of the while-loop is an array I that is a subdivision of A into sorted input buffers ready to be merged. The procedure ALLOCATE-FUNNEL creates a k-funnel on top of these buffers and returns the top-most binary merger of the funnel. The call to FILL merges the buffers, and we are done.

The procedure FILL need some space to which it can merge the elements, so a global array O_{global} of size N is needed. To avoid copying the elements of O_{global} back to A before returning from a recursive call, an implementation of FILL should use the technique of merging forth and back between two arrays in a way similar to the one used in the 4-way mergesort algorithm. Figure 6.4 illustrates the recursion tree of the algorithm, and Figure 6.5 illustrates how the elements are merged forth and back between A and the O_{global}.

The largest funnel used by FUNNELSORT is an $N^{1/d}$-funnel, which takes up approximately $cN^{(d+1)/2d}$ space. Since $cN^{(d+1)/2d} < cN$ for all $d \geq 2$, the space used by this funnel is sub-linear. In addition to this space, the algorithm uses the global array O_{global} of size N. Since funnelsort work locally on one funnel at a time, the total amount of space used by the algorithm is $O(N)$.

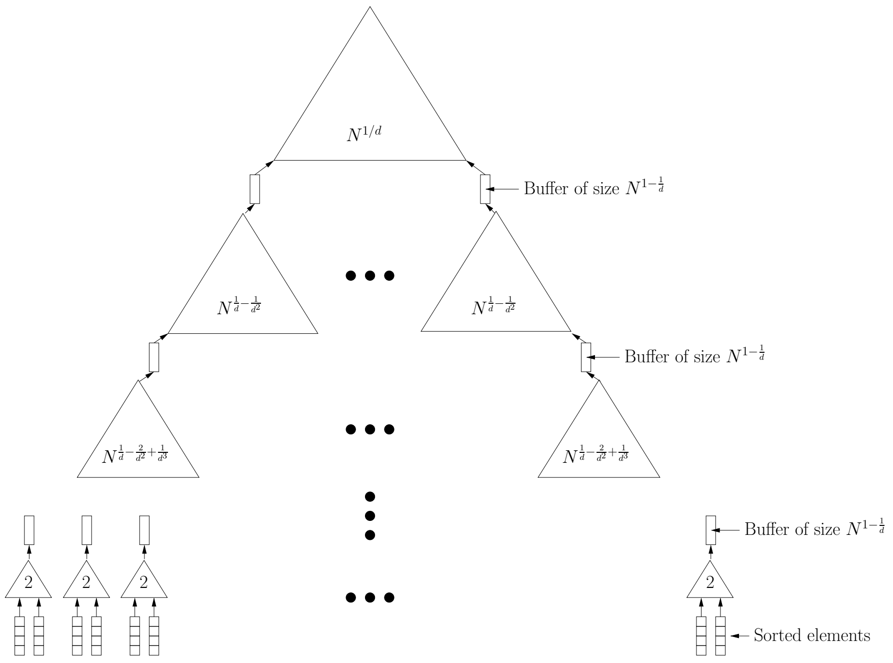

$N^{1/d}$
Buffer of size $N^{1-\frac{1}{d}}$
$N^{\frac{1}{d}-\frac{1}{d^2}}$
$N^{\frac{1}{d}-\frac{2}{d^2}+\frac{1}{d^3}}$
2
Sorted elements

Figure 6.4: The recursion tree of funnelsort. Funnelsort splits the N elements into $N^{1/d}$ segments of $N^{1-1/d}$ elements each, recursively sorts these segments using funnelsort, and merges them with an $N^{1/d}$-funnel. The bottom of the recursion is reached when a segment contains less than 2^d elements, in which case the elements are sorted using any other sorting algorithm. Each funnel merges the elements from its input streams to an output buffer that acts as input stream for the funnel just above.

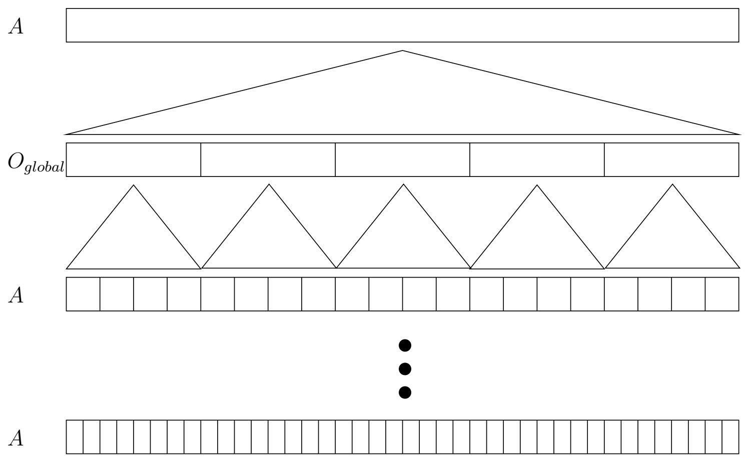


Figure 6.5: On the way up the recursion tree the elements are merged forth and back between A and O_{global}.

6.3.1 The Parameter d

In the funnelsort algorithm the parameter d determines the branching factor of the merge by influencing the number of subsequences into which funnelsort divides its input sequence. The lower we choose d, the more subsequences we get and the higher the branching factor of the merge.

A higher branching factor implies a recursion tree of fewer levels of funnels. However, this is not entirely true, because a lower value of d at the same time means that the element sequence in the base case (which is 2^d) becomes smaller. This implies a deeper recursion tree. So the parameter d offers some tradeoff between the depth of the tree of funnels and its branching factor.

However, a funnel is always a complete binary tree. So if we instead of the tree of funnels consider the entire binary merge tree of funnelsort with the funnels "unfolded", then the choice of d will have no effect on the branching factor, which will remain 2. The base case, though, will still be influenced.

It may seem tempting to tune d to obtain the best performance of funnelsort, but the fact that d determines the base case of the recursion makes it questionable to do so in a cache-oblivious context. Tuning d to obtain the best performance or complexity bound would implicitly imply the tuning of the base case to the characteristics of a specific memory system. To see how, consider a choice of $d = 8$. This choice implies that funnelsort switches to another sorting algorithm when a subsequence contains less than $2^8 = 256$ elements. Since 256 elements most likely take up the space of several cache lines, the cache complexity of the base-case sorting algorithm might compromise the good cache complexity bound of funnelsort.

To conclude, it seems reasonable to choose d, such that the base case sequence is of a reasonably small size, say $d \in \{2, 3, 4\}$[4].

6.4 Analyzing Funnelsort

The cache complexity of funnelsort depends on the cache complexity of invoking a k-funnel. And in order to deduce the cache complexity a k-funnel, we first need to know the size of a k-funnel. This is necessary since we want to know at which point in the recursion a subfunnel fits in cache.

6.4.1 Space Complexity of a k-funnel

The size of a k-funnel is the sum of the sizes of its middle buffers and the sizes of the recursive top and bottom subfunnels. Ignoring the floors and

[4]Frigo et al. use $d = 3$ in their funnelsort.

ceilings, Brodal and Fagerberg expressed this size $S(k)$ by the recurrence

$$S(k) = (k^{1/2} + 1)S(k^{1/2}) + k^{(d+1)/2}, \tag{6.2}$$

which is bounded by $ck^{(d+1)/2}$ for $c \geq 1$[12]. Note, that this recurrence does not include the size of the output buffer of the k-funnel.

Since a k-funnel is a complete tree, Recurrence (6.2) holds the implicit invariant that *all values in the sequence* $k, k^{1/2}, k^{1/4}, \ldots$ *have to be powers of 2*, i.e., k belongs to the sequence $2^{2^{\log_2 a}}$, where $a > 0$ and integer. A more precise recurrence stating the size of a k-funnel for all k that are powers of 2 is given by

$$S(k) = S(2^{\lceil \log_2 k^{1/2} \rceil}) + 2^{\lceil \log_2 k^{1/2} \rceil} S(2^{\lfloor \log_2 k^{1/2} \rfloor}) + 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^d}. \tag{6.3}$$

Note, that Recurrence (6.3) reduces to Recurrence (6.2) for those special values of k where $2^{\lfloor \log_2 k^{1/2} \rfloor} = 2^{\lceil \log_2 k^{1/2} \rceil} = k^{1/2}$.

Theorem 22 states the space used by a k-funnel by presenting a closed form of Recurrence (6.3). It only handles the situations where $d \in \{2, 3, 4\}$. In Section 6.3.1 we explained why this limitation is reasonable.

Theorem 22. *The size of a k-funnel is bounded by*

$$\begin{array}{ll} 3 \cdot 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^2} & \textit{for } d = 2, \\ 2 \cdot 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^3} & \textit{for } d = 3, \\ 2 \cdot 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^4} & \textit{for } d = 4. \end{array}$$

Proof. We solve Recurrence (6.3) by using the substitution method:
According to [12] the solution of Recurrence (6.2) has the form $ck^{(d+1)/2}$ for a constant $c \geq 1$ and $d \geq 2$, so we assume that the solution of Recurrence (6.3) has the similar form $c2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^d}$. We start by assuming that the bound holds for $2^{\lceil \log_2 k^{1/2} \rceil}$ and $2^{\lfloor \log_2 k^{1/2} \rfloor}$ (for short denoted by $k_\uparrow$ and $k_\downarrow$ respectively), that is, that

$$S(k_\uparrow) \leq c2^{\lceil \log_2 {k_\uparrow}^{1/2} \rceil} 2^{\lfloor \log_2 {k_\uparrow}^{1/2} \rfloor^d} \tag{6.4}$$

and

$$S(k_\downarrow) \leq c2^{\lceil \log_2 {k_\downarrow}^{1/2} \rceil} 2^{\lfloor \log_2 {k_\downarrow}^{1/2} \rfloor^d}. \tag{6.5}$$

Substituting (6.4) and (6.5) into Recurrence (6.3) yields

$$\begin{aligned} S(k) \leq\; & c2^{\lceil \log_2 {k_\uparrow}^{1/2} \rceil} 2^{\lfloor \log_2 {k_\uparrow}^{1/2} \rfloor^d} + \\ & 2^{\lceil \log_2 k^{1/2} \rceil} c2^{\lceil \log_2 {k_\downarrow}^{1/2} \rceil} 2^{\lfloor \log_2 {k_\downarrow}^{1/2} \rfloor^d} + \\ & 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^d} \\ \leq\; & c2{k_\uparrow}^{1/2} {k_\uparrow}^{1/2^d} + \end{aligned}$$

$$\begin{aligned}
& 2k^{1/2}c2k_{\downarrow}{}^{1/2}k_{\downarrow}{}^{1/2^{d}} + \\
& 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{d}} && (6.6)\\
= \;& c2k_{\uparrow}{}^{(d+1)/2} + \\
& 2k^{1/2}c2k_{\downarrow}{}^{(d+1)/2} + \\
& 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{d}} \\
\leq \;& c4k^{(d+1)/4} + \\
& c4k^{(d+3)/4} + \\
& 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{d}} && (6.7)\\
\leq \;& c2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{d}}. && (6.8)
\end{aligned}$$

Steps (6.6) and (6.7) are obtained by using that $2^{\lceil \log_2 a \rceil} < 2^{\log_2 a + 1} = 2a$ and $2^{\lfloor \log_2 a \rfloor} \leq 2^{\log_2 a} = a$. For any choice of $d \geq 2$ the last term of Step (6.7) will grow faster than the sum of the two first terms when k gets sufficiently large. Therefore, we can choose a value of $c \geq 1$, so that (6.8) becomes legal. If we consider the case where $d = 3$ and choose $c = 2$, then we have

$$2 \cdot 4k + 2 \cdot 4k^{6/4} + 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{3}} \leq 2 \cdot 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{3}}.$$

On Figure 6.6 we solve this inequality graphically and deduce that $c = 2$ makes Step (6.8) legal for $k > 32$ (recall, that k is always a power of 2). We have now proven that $S(k) \leq 2 \cdot 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{3}}$ for $k > 32$, so it remains to be shown that $c = 2$ is also sufficient for $k \leq 32$. By noticing that $s(2) = 2$ (i.e., 2 elements can be sorted using 2 units of space) we can show that $S(k) \leq c \cdot 2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^{d}}$ for $k \in \{2, 4, 8, 16, 32\}$:

$$\begin{aligned}
s(2) &= 2 \\
&\leq 2 \cdot 2^{\lceil \log_2 2^{1/2} \rceil} 2^{\lfloor \log_2 2^{1/2} \rfloor^{3}} = 4 \\
s(4) &= s(2^{\lceil \log_2 4^{1/2} \rceil}) + 2^{\lceil \log_2 4^{1/2} \rceil} s(2^{\lfloor \log_2 4^{1/2} \rfloor}) + 2^{\lceil \log_2 4^{1/2} \rceil} 2^{\lfloor \log_2 4^{1/2} \rfloor^{3}} \\
&= s(2) + 2s(2) + 16 = 22 \\
&\leq 2 \cdot 2^{\lceil \log_2 4^{1/2} \rceil} 2^{\lfloor \log_2 4^{1/2} \rfloor^{3}} = 32 \\
s(8) &= s(2^{\lceil \log_2 8^{1/2} \rceil}) + 2^{\lceil \log_2 8^{1/2} \rceil} s(2^{\lfloor \log_2 8^{1/2} \rfloor}) + 2^{\lceil \log_2 8^{1/2} \rceil} 2^{\lfloor \log_2 8^{1/2} \rfloor^{3}} \\
&= s(4) + 4s(2) + 32 = 62 \\
&\leq 2 \cdot 2^{\lceil \log_2 8^{1/2} \rceil} 2^{\lfloor \log_2 8^{1/2} \rfloor^{3}} = 64 \\
s(16) &= s(2^{\lceil \log_2 16^{1/2} \rceil}) + 2^{\lceil \log_2 16^{1/2} \rceil} s(2^{\lfloor \log_2 16^{1/2} \rfloor}) + 2^{\lceil \log_2 16^{1/2} \rceil} 2^{\lfloor \log_2 16^{1/2} \rfloor^{3}} \\
&= s(4) + 4s(4) + 256 = 366 \\
&\leq 2 \cdot 2^{\lceil \log_2 16^{1/2} \rceil} 2^{\lfloor \log_2 16^{1/2} \rfloor^{3}} = 512 \\
s(32) &= s(2^{\lceil \log_2 32^{1/2} \rceil}) + 2^{\lceil \log_2 32^{1/2} \rceil} s(2^{\lfloor \log_2 32^{1/2} \rfloor}) + 2^{\lceil \log_2 32^{1/2} \rceil} 2^{\lfloor \log_2 32^{1/2} \rfloor^{3}}
\end{aligned}$$

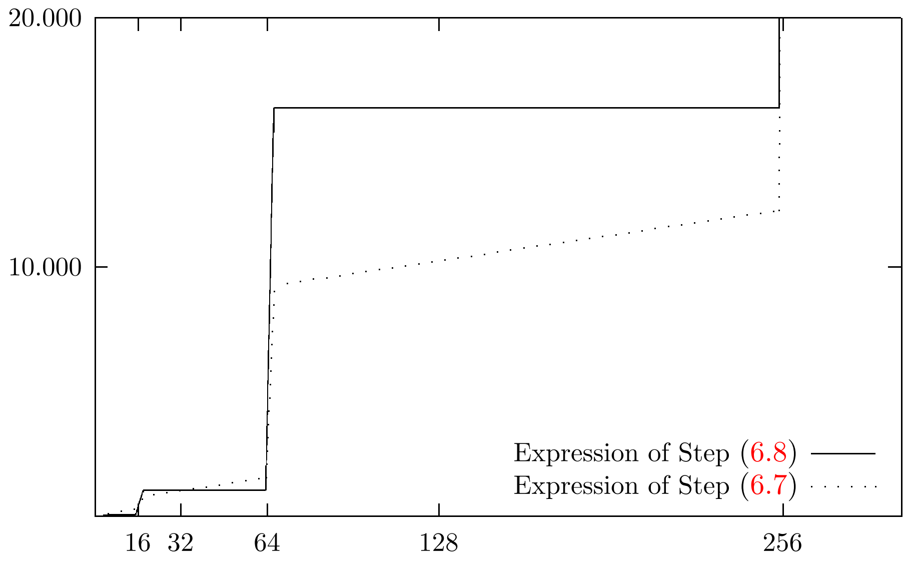


Figure 6.6: For the values $d = 3$ and $c = 2$ Step (6.8) is legal when $k > 32$.

$$
\begin{aligned}
&= s(8) + 8s(4) + 512 = 750 \\
&\leq 2 \cdot 2^{\lceil \log_2 32^{1/2} \rceil} 2^{\lfloor \log_2 32^{1/2} \rfloor^3} = 1024.
\end{aligned}
$$

In a similar way it can be shown that $c = 3$ is sufficient for $d = 2$, and that $c = 2$ is sufficient for $d = 4$. This concludes the proof of Theorem 22. $\square$

6.4.2 Cache Complexity of a k-funnel

Now that we know how much space a k-funnel occupies we can analyze its cache complexity. In Theorem 23, that states the cache complexity of a single invocation of a k-funnel, we will ignore the floors and ceilings of the space complexity bound derived in Theorem 22 and use the bound given by Brodal & Fagerberg instead. We do this to simplify the analysis, and since their bound $c'k^{(d+1)/2}$ is an upper bound for $c2^{\lceil \log_2 k^{1/2} \rceil} 2^{\lfloor \log_2 k^{1/2} \rfloor^d}$, when $c' \geq 2c$, this will not affect the cache complexity bound in the wrong direction, that is, making it look better than it actually is.

In Theorem 23 we use the following variant of the tall-cache assumption:

$$B^{(d+1)/(d-1)} \leq M/2c' \tag{6.9}$$

By checking for the values of c and d we derived in theorem 22 it is easy to see that (6.9) still describes a tall cache.

Theorem 23. *Assuming a tall cache (Inequality (6.9)) and that the size of a k-funnel is bounded by $c'k^{(d+1)/2}$ for a constant $c' \geq 1$ and $d \geq 2$, a single invocation of a k-funnel causes $Q(N) \leq \left(1+\frac{1}{d}\right)(2c'+5)\frac{k^d}{B}\log_M k^d + k$ cache misses.*

Proof. To prove this bound we consider the recursive definition of the buffer sizes in a k-funnel. We find the point in the recursion where a subfunnel (top or bottom) for the first time occupies at most half of the cache. To be precise, we consider the point of recursion where the size of the subfunnel satisfies the inequality $c'\bar{k}^{(d+1)/2} \leq M/2$, where $\bar{k}$ denotes the number of leaves of the subfunnel. This implies

$$\bar{k}^{(d+1)/2} \leq M/2c', \tag{6.10}$$

and hence

$$\bar{k} \leq (M/2c')^{2/(d+1)}. \tag{6.11}$$

Note, that since $\bar{k}$ is the first such value of k we also know that

$$(\bar{k}^2)^{(d+1)/2} = \bar{k}^{d+1} > M/2c', \tag{6.12}$$

which implies

$$\bar{k} > (M/2c')^{1/(d+1)}. \tag{6.13}$$

We denote subfunnels that satisfy Inequality (6.10) as *base* funnels, and we denote the output buffers of base funnels as *large* buffers.

By Inequality (6.11) and the tall-cache assumption (6.9) a base funnel and one block of each of its $\bar{k}$ leaf input streams can at the same time fit in the cache, since

$$\bar{k} \cdot B \leq (M/2c')^{2/(d+1)} \cdot (M/2c')^{(d-1)/(d+1)} \leq M/2c'.$$

Loading a base funnel and one block of each of its $\bar{k}$ input streams incurs

$$\frac{c\bar{k}^{(d+1)/2}}{B} + \bar{k} \tag{6.14}$$

cache misses. Expression (6.14) defines the *load cost* of a base funnel.

Now, consider the number of cache misses incurred by a single invocation of a base funnel that outputs $\bar{k}^d$ elements to its output buffer. Since the entire base funnel and one block for each of its $\bar{k}$ input buffers can be in the cache simultaneously, the invocation incurs

$$\frac{2\bar{k}^d}{B} + \bar{k} \tag{6.15}$$

cache misses. That is, $\bar{k}^d/B$ cache misses for outputting the elements at the root of the base funnel, $\bar{k}^d/B$ cache misses for reading the elements from the input streams, and an additional $\bar{k}$ misses, because the elements of the $\bar{k}$ input streams may be aligned badly. Expression (6.15) defines the *merge cost* of filling a large buffer.

In the following argumentation we assume that each of the k input streams of the k-funnel contribute to the merge with an equal amount of elements. This assumption is reasonable, since it reflects the way in which k-funnels are used in funnelsort. A consequence of this is, that during a call FILL to the root of the k-funnel, an equal number of elements passes through all large buffers that belong to the same layer of the k-funnel.

A base funnel has at least $(M/2c')^{1/(d+1)}$ leaves (cf., Inequality (6.13)), so the path from the root to a leaf in the k-funnel consists of

$$\begin{aligned}
\log_{(M/2c')^{1/(d+1)}} k &= \frac{\log_{(M/2c')} k}{\log_{(M/2c')}(M/2c')^{1/(d+1)}} \\
&= \log_{(M/2c')} k^{d+1} \\
&= \frac{\log_M k^{d+1}}{\log_M(M/2c')} \\
&= \frac{1}{1-\log_M(2c')} \log_M k^{d+1}
\end{aligned}$$

base funnels, and equally many large buffers — including the output buffer of the k-funnel itself. Since $\log_M(2c')$ is very small, the k-funnel has approximately $\log_M k^{d+1}$ levels of large buffers. Figure 6.7 illustrates this. Merging the k^d elements through one of these $\log_M k^{d+1}$ layers corresponds to filling $k^d \bar{k}^d$ large buffers. Since the merge cost of filling a large buffer is $2\bar{k}^d/B + \bar{k}$ cache misses, the cost of merging k^d elements with the k-funnel becomes

$$\left(\frac{2\bar{k}^d}{B} + \bar{k}\right) \frac{k^d}{\bar{k}^d} \log_M k^{d+1} \tag{6.16}$$

cache misses.

In addition to this total merge cost, we also have to pay for the swapping of base funnels in and out of the cache, as the FILL algorithm jumps around in the k-funnel working on filling the different large buffers. To derive the total cost of this swapping, consider a call FILL to the root of a base funnel F, which outputs $\bar{k}^d$ elements to the output buffer of F. During the call, an input buffer of F may run empty. In order to refill this empty large buffer, a call FILL to the root of the base funnel just below the empty buffer is triggered. During this recursive filling of the empty buffer, F is most likely evicted completely from the cache, which leads to its reload when the empty buffer has been refilled. By Expression (6.14) the cost of reloading the base funnel is $\frac{c'\bar{k}^{(d+1)/2}}{B} + \bar{k}$ cache misses, so in the worst case we have to pay twice the load cost each time a large buffer is filled; one payment when the filling

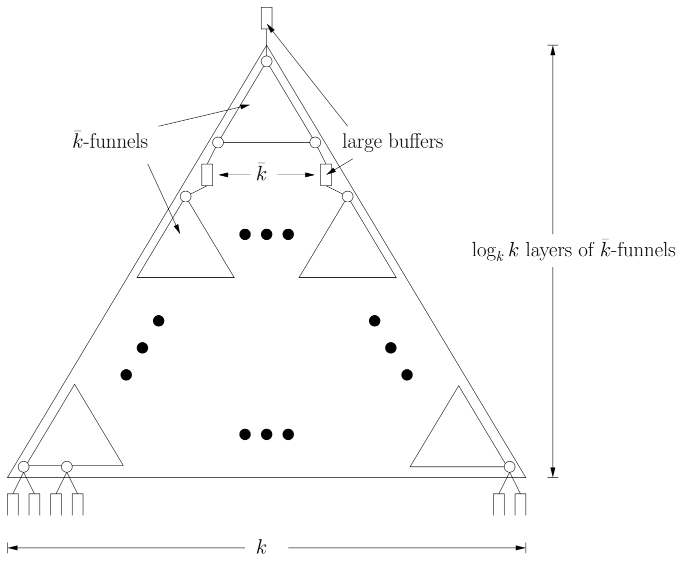


Figure 6.7: The path from the root to a leaf in a k-funnel consists of approximately $\log_M k^{d+1}$ base funnels and equally many large buffers, if including the output buffer of the entire k-funnel.

of the buffer begins, and one additional payment during the fill. The total load cost incurred by a single invocation of a k-funnel becomes

$$\left(\frac{2c'\bar{k}^{(d+1)/2}}{B} + 2\bar{k} \right) \frac{k^d}{\bar{k}^d} \log_M k^{d+1} \tag{6.17}$$

cache misses.

The input buffers get exhausted at some point during the merge, which means that the buffer will not contribute with any more elements to the merge. The exhaustion of the buffers is propagated as far upward the k-funnel as possible. This means that a buffer is marked as exhausted when both its input buffers are exhausted, and that the first buffers that get exhausted therefore are the input streams of the k-funnel.

The removal of the final element of an input stream may complete the filling of an output buffer of a $\bar{k}$-funnel. If this is the case, then the k-funnel does not know that the stream is exhausted until the next time it is invoked. During this next invocation the $\bar{k}$-funnel will therefore access the exhausted stream. This process will incur a cache miss that is accounted for by neither the merge cost nor the load cost. In the worst case this exhaustion cost is paid for all input streams, which adds in total an additional k cache misses to the merge and load costs of the k-funnel.

Summing the exhaustion cost, the merge cost, and the load cost, a single invocation of a k-funnel outputting k^d elements incurs in total

$$\begin{aligned}
& \left(\frac{2c'\bar{k}^{(d+1)/2}}{B} + \frac{2\bar{k}^d}{B} + 3\bar{k}\right)\frac{k^d}{\bar{k}^d}\log_M k^{d+1} + k \\
< \; & \left(\frac{2c'\bar{k}^d}{B} + \frac{2\bar{k}^d}{B} + 3\bar{k}\right)\frac{k^d}{\bar{k}^d}\log_M k^{d+1} + k && (6.18) \\
= \; & \left(\frac{(2c'+2)k^d}{B} + \frac{3k^d}{\bar{k}^{d-1}}\right)\log_M k^{d+1} + k \\
\leq \; & (2c'+5)\frac{k^d}{B}\log_M k^{d+1} + k && (6.19) \\
= \; & \left(1 + \frac{1}{d}\right)(2c'+5)\frac{k^d}{B}\log_M k^d + k
\end{aligned}$$

cache misses in the worst case. In Step (6.18) we use that $d \geq 2$, which implies $\bar{k}^{(d+1)/2} < \bar{k}^d$, and in Step (6.19) we use Inequality (6.12), which implies $\bar{k}^{d-1} > (M/2c')^{(d-1)/(d+1)} \geq B$. This concludes the proof of Theorem 23. $\square$

6.4.3 Cache Complexity of Funnelsort

The following theorem states the cache complexity of funnelsort.

Theorem 24. *Assuming a tall cache as defined in Inequality (6.9), the number of cache misses incurred by funnelsort is* $Q(N) \leq \left(1 + \frac{1}{d}\right)(2c' + 5)d\frac{N}{B}\log_M N$.

Proof. Funnelsort recursively sorts $N^{1/d}$ segments of $N^{1-1/d}$ elements and merges these with a $N^{1/d}$-funnel. Each of the $N^{1/d}$ segments are sorted recursively using smaller and smaller funnels, so the $N^{1/d}$-funnel at the coarsest level of recursion is the biggest funnel used by the algorithm. The size of the $N^{1/d}$-funnel is bounded by $c'N^{(d+1)/2d} < c'N$, because $d \geq 2$.

Consider the point in the recursion where the size of the funnel is smaller than half the cache for the first time, i.e., when $N < M/2$. In the cache-complexity analysis this is considered the base case of the recursion[5]. In the base case the $N^{1/d}$-funnel can be contained entirely in the cache, and the N elements can therefore be merged cheaply. By Inequalities (6.14) and (6.15) loading the funnel into the cache incurs $\frac{c'N^{(d+1)/2d}}{B} + N^{1/d}$ cache misses, and merging the elements incurs an additional $2\frac{N}{B} + N^{1/d}$ cache misses. In total

[5]This is different from the algorithmic base case that is reached when $N < 2^d$.

this is

$$\begin{aligned} & \frac{c' N^{(d+1)/2d}}{B} + 2\frac{N}{B} + 2N^{1/d} \\ \leq \;& (c'+2)\frac{N}{B} + 2N^{1/d} \\ \leq \;& (c'+5)\frac{N}{B}. \end{aligned}$$

cache misses, which is clearly bounded by $\left(1+\frac{1}{d}\right)(2c'+5)d\frac{N}{B}\log_M N$ as stated. The last derivation depends on $\frac{N}{B}$ being an upper bound for $N^{1/d}$. To see why it is so, consider the recursive definition of a base funnel of Section 6.4.2. Recall, that a base funnel (i.e., a $\bar{k}$-funnel) is the first subfunnel of a k-funnel that takes up at most half the cache (cf., Inequality (6.10)). This implies that the subfunnel one step further up the recursion tree is larger than half the cache size. By Inequality (6.12) we have $\bar{k}^{d+1} > M/2c$, which implies $\bar{k}^{d-1} > (M/2c')^{(d+1)/(d-1)} \geq B$ and hence $\bar{k} \leq \frac{\bar{k}^d}{B}$. In analogy to this we can deduce that $N^{1/d} \leq \frac{N}{B}$.

When $N \geq M/2$ the cache complexity of funnelsort is given by the recurrence

$$Q(N) = N^{1/d}Q(N^{1-1/d}) + \left(1+\frac{1}{d}\right)(2c'+5)\frac{N}{B}\log_M N + N^{1/d}, \qquad (6.20)$$

since the funnelsort algorithm guarantees that only one funnel is active at a time.

In solving this recurrence we use $\alpha = \left(1+\frac{1}{d}\right)(2c'+5)$ for notational convenience. Assuming that the bound $Q(N) \leq \alpha d\frac{N}{B}\log_M N - b$ holds for $N^{1-1/d}$ and substituting into the recurrence yields

$$\begin{aligned} Q(N) \leq \;& N^{1/d}\left(\alpha d\frac{N^{1-1/d}}{B}\log_M N^{1-1/d} - b\right) + \alpha\frac{N}{B}\log_M N + N^{1/d} \\ = \;& \alpha d\frac{N}{B}\log_M \frac{N}{N^{1/d}} - bN^{1/d} + \alpha d\frac{N}{B}\log_M N^{1/d} + N^{1/d} \\ = \;& \alpha d\frac{N}{B}\log_M N - \alpha d\frac{N}{B}\log_M N^{1/d} - bN^{1/d} + \alpha d\frac{N}{B}\log_M N^{1/d} + N^{1/d} \\ = \;& \alpha d\frac{N}{B}\log_M N - bN^{1/d} + N^{1/d} \\ \leq \;& \alpha d\frac{N}{B}\log_M N - b, \end{aligned}$$

when $b > 1$, because $N^{1/d} \geq 1$. This concludes the proof of Theorem 24 □

6.4.4 Work Complexity of Funnelsort

To deduce the work complexity of funnelsort, we consider the algorithm FUNNELSORT on page 92. The algorithm first divides the problem into $N^{1/d}$

recursive subproblems each of size $N^{1-1/d}$. Then it creates a $N^{1/d}$-funnel, and invokes this funnel once to merge the subproblems. The work complexity $W(N)$ of Funnelsort is described by the recurrence

$$W(N) = \begin{cases} W_{base}(N) & \text{if } N < 2^d, \\ N^{1/d}W(N^{1-1/d}) + & \\ \quad W_c(N^{1/d}) + W_f(N) + O(N) & \text{otherwise.} \end{cases} \tag{6.21}$$

W_{base} is the work complexity of the sorting algorithm used for the base case of funnelsort, the $W_c(N^{1/d})$ term is the work complexity of building an $N^{1/d}$-funnel with the Allocate-Funnel-Rec procedure (see page 90), and the $W_f(N)$ term is the work complexity of filling the output buffer of this $N^{1/d}$-funnel with N elements using the Fill procedure (see page 91).

Assuming that the procedure Allocate-Buffer takes constant time, the work complexity of the Allocate-Funnel-Rec is given by the recurrence

$$W_c(N) = (N^{1/2} + 1)W_c(N^{1/2}) + O(N).$$

The solution to this recurrence is $O(N \log_2 \log_2 N)$, so in Recurrence (6.21) we have $W_c(N^{1/d}) = O(N^{1/d} \log_2 N^{1/d})$, which means that the cost of building the funnel is dominated by the $O(N)$ term of (6.21).

For the W_f term we examine the call Fill to the root merger of the $N^{1/d}$-funnel. This call moves each of the N elements from a leaf buffer to the output buffer of the $N^{1/d}$-funnel.

Fill calls itself recursively each time a buffer becomes empty. But since the buffers within the $N^{1/d}$-funnel are of different sizes, not all buffers become empty equally many times. Therefore, it is difficult to describe the total number of recursive calls by a recurrence.

Instead, we notice that the $N^{1/d}$-funnel is in fact in complete binary tree of $2N^{1/d} - 1$ nodes, which means that the funnel consists $\log_2(2N^{1/d} - 1)$ buffer levels. Each of the N elements passes through each buffer level on the leaf-to-root path, from which it follows that a call Fill to the root of the $N^{1/d}$-funnel incurs

$$\begin{aligned} & N \log_2(2N^{1/d} - 1) \\ < \quad & N \log_2(2N^{1/d}) \\ = \quad & N + \frac{1}{d} N \log_2 N \end{aligned} \tag{6.22}$$

element moves, and hence, equally many element comparisons and checks for empty buffers. Since a single buffer can contain many elements, the number of buffers that becomes empty during the call is also bounded from above by Expression (6.22).

Program 13 is a pure-C translation of the Fill procedure. By use of this program we can count the number of pure-C instructions that are executed

```
void fill() {
  bool a1, a2, b1, b2, c1, c2;
  T x, y;
  bool exhausted1 = false;
  bool exhausted2 = false;
 loop:
  a1 = out->full();                     // 1 pure-C instruction
  if (a1 == true) goto out;
  a1 = !in1->empty();                   // 1 pure-C instruction
  if (exhausted1 || a1) goto cont1;
  a1 = !source1->exhausted();           // 1 pure-C instruction
  if (a1) goto end1;
  exhausted1 = true;
  goto cont1;
 end1:
  source1->fill();
 cont1:
  a1 = !in2->empty();                   // 1 pure-C instruction
  if (exhausted2 || a1) goto cont2;
  a1 = !source2->exhausted();           // 1 pure-C instruction
  if (a1) goto end2;
  exhausted2 = true;
  goto cont2;
 end2:
  source2->fill();
 cont2:
  if (exhausted1 && exhausted2) goto out;
  if (!exhausted1) goto cont3;
  x = in2->extract();                   // 6 pure-C instructions
  out->insert(x);                       // 6 pure-C instructions
  goto loop;
 cont3:
  if (!exhausted2) goto cont4;
  x = in1->extract();                   // 6 pure-C instructions
  out->insert(x);                       // 6 pure-C instructions
  goto loop;
 cont4:
  x = in1->peep();                      // 1 pure-C instruction
  y = in2->peep();                      // 1 pure-C instruction
  if (comp(x, y)) goto cont5;           // 1 pure-C instruction
  x = in2->extract();                   // 6 pure-C instructions
  out->insert(x);                       // 6 pure-C instructions
  goto loop;
 cont5:
  x = in1->extract();                   // 6 pure-C instructions
  out->insert(x);                       // 6 pure-C instructions
  goto loop;
 out:
  return;
}
```

Program 13: Pure-C implementation of the FILL procedure. *in*1, *in*2, and *out* are the input and output buffers of the binary merger, and *source*1 and *source*2 are the binary mergers just below the merger that the procedure is invoked on.

in the worst case for each element moved. In the program, `in1`, `in2`, and `out` are pointers to the input and output buffers of the binary-merger object on which `fill()` is called. `source1` and `source2` are pointers to the binary-merger objects at the other end of the input buffers. The program contains calls to some functions that manipulate these buffer objects and binary-merger objects. The Pure-C cost of these calls appears from the program.

The while-loop (line 6–47) results in an element move in two situations. Either when exactly one of the input buffers is empty and the source node below it is exhausted, or when none of the input buffers are empty. In all other situations the while-loop results in a recursive call to `fill()` on a binary-merger object further down the merge tree.

If `in1` is empty and `source1` is exhausted, then the element move incurs 27 pure-C instructions, since the lines 4–14, 18–19, and 27–31 are executed. If `in2` is empty and `source2` is exhausted, then the move incurs 28 pure-C instructions, since the lines 4–10, 18–23, 27–28, and 33-36 are executed. If neither `in1` nor `in2` is exhausted, then the lines 4–10, 18–19, 27–28, 33, and either 38–43 or 38–40 and 45–47 are executed, which incurs 27 pure-C instructions. By Expression (6.22) this results in a worst-case pure-C instruction count of $28\frac{1}{d}N\log_2 N + O(N)$, for a call to `fill()`, which means that the work complexity of funnelsort in terms of pure-C instructions is given by the recurrence

$$W_p(N) \leq \begin{cases} O(N\log_2 N) & \text{if } N < 2^d, \\ N^{1/d}W_p(N^{1-1/d}) + \frac{28}{d}N\log_2 N + O(N) & \text{otherwise.} \end{cases} \tag{6.23}$$

The following theorem states the pure-C instruction count of funnelsort.

Theorem 25. *If the* Fill *procedure is implemented as in Program 13, then funnelsort incurs at most $28N\log_2 N + O(N)$ pure-C instructions when sorting N elements.*

Proof. For the recursive case of Recurrence (6.23) we assume that the bound holds for $N^{1-1/d}$, that is $W_p(N^{1-1/d}) \leq 28N^{1-1/d}\log_2 N^{1-1/d} + O(N^{1-1/d})$. Substituting $W_p(N^{1-1/d})$ into the recurrence yields

$$\begin{aligned} W_p &\leq N^{1/d}(28N^{1-1/d}\log_2 N^{1-1/d} + O(N^{1-1/d})) + \frac{28}{d}N\log_2 N + O(N) \\ &= 28N\log_2 N^{1-1/d} + O(N) + \frac{28}{d}N\log_2 N + O(N) \\ &= (1-1/d)28N\log_2 N + \frac{28}{d}N\log_2 N + O(N) \\ &= 28N\log_2 N - \frac{28}{d}N\log_2 N + \frac{28}{d}N\log_2 N + O(N) \\ &= 28N\log_2 N + O(N). \end{aligned}$$

Algorithm	Work complexity	Cache complexity
4-way mergesort	$3.25N \log_2 N + O(N)$	$\approx 0.0625N \log_2 N$
Funnelsort ($d = 2$)	$24N \log_2 N + O(N)$	$\approx 0.20N \log_2 N$
Funnelsort ($d = 3$)	$24N \log_2 N + O(N)$	$\approx 0.20N \log_2 N$
Funnelsort ($d = 4$)	$24N \log_2 N + O(N)$	$\approx 0.25N \log_2 N$

Figure 6.8: The work and cache complexities of the sorting algorithms. The logarithms are rewritten into base 2 to make the comparison easier.

The bound also holds for the base case as long as we choose a base-case sorting algorithm that incurs at most $28N \log_2 N + O(N)$ pure-C instructions. This concludes the proof of Theorem 25. □

6.5 Expected Performance

We have summarized the cache and work complexities of the 4-way mergesort and funnelsort in Figure 6.8. To ease the comparison, all logarithms have been changed into base 2, and the characteristics of the level 2 cache of the Athlon computer has been assumed ($M = 65536$ and $B = 16$ for 4-byte elements). For the cache complexity of the 4-way mergesort we have only shown the most significant term, and for the cache complexity of funnelsort, we have used the values of c derived in Theorem 22 on page 95.

The constant in the work complexity of funnelsort is approximately 8 times larger then that of 4-way mergesort. This is not surprising, since it reflects the fact that the merging step in funnelsort is quite complex.

It is more surprising that the cache complexity of funnelsort is between 3 and 4 times worse than that of 4-way mergesort. This huge difference can partially be explained by the fact that the analysis is of worst-case performance. However, some of the difference can be explained by the different ways in which we have conducted the cache complexity analyses.

6.5.1 The Accuracy of the Analysis

In the cache complexity analysis of the 4-way mergesort we assumed that we could keep the first block of the last 3 subsequences in cache throughout an entire merge step in order to save some cache misses. In the funnelsort analysis we did not use that trick.

If we had used the similar assumption that, at any time during the merge, the cache could hold a $\bar{k}$-funnel, the first block of each of the $\bar{k}$ input buffers, and the first block of the last $\bar{k} - 1$ input buffers simultaneously, then we could have saved $\bar{k} - 1$ cache misses per invocation of a base funnel.

Expression (6.15) (the merge cost) could have been rewritten into

$$\frac{2\bar{k}^d}{B} + 1. \tag{6.24}$$

This would have lead to a cache complexity of

$$Q(N) \leq \left(1 + \frac{1}{d}\right)(2c' + 4)d\frac{N}{B}\log_M N,$$

which is a little better than the one derived in Theorem 24. Note that this additional assumption does not influence the load cost of the $\bar{k}$-funnel (Equation (6.14)), since the $\bar{k} - 1$ elements that should remain in the cache during the merge are a subset of the $\bar{k}$ elements that are already counted in the load cost.

In the cache complexity analysis of a k-funnel, we furthermore used a somewhat pessimistic bound on its space complexity (see Section 6.4.2 on page 97). We did this in order to simplify the analysis, but the tradeoff was a doubling of the constant factor c of the space complexity bound, which may be more than needed. Without this doubling the funnelsort would have incurred $0.13N \log_2 N$ and not $0.20N \log_2 N$ cache misses in Figure 6.8 for $d = 3$.

To conclude, a more careful analysis than the one we have conducted might have lead to a tighter worst-case cache complexity bound on funnelsort. We therefore expect that funnelsort will exhibit better cache usage than predicted in this chapter in the average-case.

6.6 Summary

In this Chapter we have derived the constant factors of the cache complexity of 4-way mergesort and funnelsort. Furthermore, we have described the workings of the k-funnel and the funnelsort algorithm in detail. In order to derive the pure-C instruction count of the funnelsort algorithm, we have translated the central part of the algorithm into pure-C.

CHAPTER 7

Benchmarks

An experiment is a question which science poses to Nature, and a measurement is the recording of Nature's answer.

— Max Planck (1949)

Most often the goal of benchmarking computer programs is to obtain empirical results that support some theoretically founded hypothesis stating that, program A is faster than program B, or program A uses only half as much memory as program B.

No matter which hypothesis one has the intention of supporting — or rejecting — it is important that the measurements are both reliable and valid. A measurement is *reliable* if comparable but independent measurements of the same program agree. For example, a reliable measurement can be obtained by conducting the same measurements multiple times and on different computers. A measurement is *valid* if it in fact tells us something about what we set out to investigate. For example, measuring the execution time of a program does not directly tell us anything about the way in which the memory hierarchy affects the behavior of the program, but combining execution time measurements with the number of cache misses incurred by the program certainly will.

As one can imagine, benchmarking computer programs is not an easy task. Numerous conditions should be taken into account in order to obtain useful results. In this chapter we first describe our benchmarking methodology (Section 7.1). Thereafter, we investigate the benchmarking results of the static search tree implementations (Section 7.2), and of the sorting implementations (Section 7.3).

7.1 Methodology

Preparing and running benchmarks can be viewed as the task of doing a performance investigation. Which aspects of a program that we expect the investigation to tell something about highly influences how the benchmarks should be constructed. Therefore, we first of all have to define the goal of the investigation.

7.1.1 What Should Be Measured

In this thesis we are interested in obtaining empirical measurements that can help us in answering the following questions:

1. Is the cache-oblivious approach successful when it comes to applying it to searching and sorting algorithms? And if it is, under which conditions is the approach most and least advantageous?
2. Do empirical measurements support our analytical predictions of the cache complexities of the algorithms? And if not, how and and why do they differ?
3. Do empirical measurements support our analytical predictions of the work complexities and branch misprediction counts of the algorithms? And if not, can we explain the differences from our use of cost model or is there another explanation?

To answer these questions for each the implemented programs, we need to measure the running time, the way in which the programs uses the memory system, the instruction count, and the branch misprediction count.

Measuring Execution Time

The execution time of a program is usually measured in either wall-clock time or CPU time. Wall-clock time corresponds to starting a stop-watch when the program starts executing, and stopping the watch when the program terminates, while CPU time is the time the CPU uses working on the program. The advantage of using CPU time is that the measurements will not be affected by interrupts or other processes running on the system. On the contrary, wall-clock time will measure all activities on the system from the benchmarked program begins its execution until it terminates. The consequence of using wall-clock time is therefore a low reliability of the measurements. That is, the result may be measurements that are highly dependent on the setup of the benchmarking computer at the time when the benchmarks were run, and therefore hard to reproduce and generalize. Nevertheless, there is one important objection to using CPU time as the sole execution time metric, namely that the CPU cannot execute the instructions of a process while it waits for main memory. The operating system will simply switch to another process or stay idle while waiting, so the measured CPU time will not reflect the true influence of the memory system on execution time. The consequence is that we cannot use CPU time to measure the effects of datasets that exceeds main memory in size.

To get the best of two worlds, we choose CPU time as a metric for datasets that are smaller than main memory and wall-clock time for the datasets that exceeds it in size.

Measuring Memory System Performance

PAPI [36] is an acronym for Performance Application Programming Interface, and can be used to access the hardware performance counters found on most modern CPUs. By counting events such as floating point operations and hardware interrupts PAPI can measure a wide variety of performance characteristics. The use of hardware parameters ensures that monitoring the performance of a program happens without affecting its behavior.

Through a C++ interface we can use PAPI to monitor events regarding the memory system that happen during program execution. We are interested in monitoring accesses to the various cache layers and in particular counting cache misses. The PAPI events of interest are shown in the following table.

PAPI event	Description
`PAPI_L1_DCA`	Level 1 Data Cache Accesses
`PAPI_L1_DCM`	Level 1 Data Cache Misses
`PAPI_L2_DCA`	Level 2 Data Cache Accesses
`PAPI_L2_DCM`	Level 2 Data Cache Misses

In practice, we instruct the hardware counters to monitor these PAPI events. By reading the counters before and after each benchmark case, we obtain the various counts.

Unfortunately, PAPI cannot be used to monitor page faults, as paging is handled by the operating system. We therefore need some method to get hold of the page fault count through the operating system. In the Linux operating system the `/proc` file system can be used to access execution statistics of the processes running on the system at any time. Each running process has a subdirectory in `/proc` named by its process-id, and through the files in that directory we can access information on the environment in which the process executes, the currently mapped memory regions of the process, and also the number of page faults incurred by the process. This information on page faults can be read from the file `stat`.

According to the manual page for the `/proc` file system there are two types of page faults. A *minor* page fault happens when a new page is created, and therefore does not incur disk access[1]. A *major* page fault happens when a page is requested that is not in main memory. Major page faults therefore require disk reads, so this is the type we will monitor.

Counting Instructions

Analytically, we have expressed the work complexities of the various algorithms by pure-C instruction counts. By monitoring the PAPI event `PAPI_TOT_INS` we can count the number of instructions that the CPU actually executes while running the programs. By comparing these two measures we might get an idea of the quality of the derived work complexities.

Counting Branch Misprediction

Also branch mispredictions can be measured by PAPI. We do this by monitoring the event `PAPI_BR_MSP`. In order to derive the percentage of mispredicted conditional branch instructions, we also monitor the total number of conditional branch instructions executed, that is, the event `PAPI_BR_CN`.

7.1.2 Benchmarking Platforms

Ideally, the benchmarks should be carried out on a variety of different computers representing the various CPU and computer manufacturers, and thereby the various trends in modern CPUs and computer design. This should be done in order to obtain reliable and generalizable results as well as to investigate how architectural details might affect performance.

Our benchmark computers (see Figure 2.5) only represent two CPU manufacturers. However, the Pentium® 4 and the Athlon are among the most recent CPUs from Intel® and AMD aimed at the marked of personal computers, and they are both widely used. Nevertheless, we remark that the benchmark results cannot necessarily be generalized to other platforms.

Unfortunately, PAPI only works on the Pentium® 4 CPU in a preliminary alpha version. To optimize the use of the computers we therefore decide on measuring the PAPI event counts on the Athlon while we use the Pentium for benchmarking large datasets.

The benchmarking computers are fully at our disposal, so we have control of all processes running on them. In order to make the wall-clock time measurements as reliable as possible, we have shut down all irrelevant processes prior to running the benchmark cases.

[1]Minor page faults are sometimes referred to as page *reclaims*.

7.1.3 Benchmarking Tool

For benchmarking we use a tool originally written by Jyrki Katajainen for the CPH STL Project[2]. By using a benchmarking tool we can automate the execution of the benchmarks, and ensure that all benchmark cases are carried out uniformly. That is, the code that performs the various types of measurements is identical for all implementations.

The version of the tool that we use[3] contains a driver for measuring CPU time. For each benchmark case this driver measures 20 runs and returns the median of these 20 execution times. However, if more than 10 percent of the runs differ by more than 20 percent from the median, then an additional 20 runs are measured and the check is done again on all the runs (i.e., 40 runs). This procedure is continued 100 times or until the median is considered reliable.

We have written a couple of additional drivers for measuring PAPI events, wall-clock time, and page faults. The wall-clock and PAPI drivers use the same procedure of evaluating the reliability of the measurements, but for the PAPI driver we occasionally had to lower the reliability a little by accepting the median when at most 30 percent of the runs differed from it by more than 20 percent.

Due to the fact that page faults take quite some time to process, the page fault benchmarks may run for a very long time. This makes it practically impossible to use the median of 20 runs. However, it is our experience that the page fault counts do not differ significantly between runs, so we find it acceptable to let each page fault benchmark case consist of only 5 runs where we use the median.

The benchmark tool offers the possibility of carrying out the measurements by use of the so-called *double-loop* technique. By this technique each benchmark is followed by an identical benchmark performed with empty functions. By subtracting the result of the empty benchmark from that of the real run, the overhead imposed by the benchmark system itself can be eliminated. To use the double-loop technique for the static search layout policies we have written an empty layout policy.

The benchmarking tool is set up to use `g++` — the GNU C++ compiler. Furthermore, all implementations are compiled with the `-O3` option, that is, the highest level of optimization.

7.1.4 Datasets Sizes

When measuring execution times, and investigating the connection between execution time and the number of cache misses that a program incurs, some input data sizes become more interesting than others. In particular, it is

[2]http://www.cphstl.dk
[3]Version 1.6 dated January 26th, 2003.

Program	Athlon		Pentium		Main
	L1	L2	L1	L2	Memory
Inorder	16.384	65.536	2.048	131.072	8.388.608
Imp. Heap	16.384	65.536	2.048	131.072	8.388.608
Exp. Heap	7.168	28.672	896	57.344	3.670.016
Imp. Height Part.	16.384	65.536	2048	131.072	8.388.608
Exp. Height Part.	5.461	21.845	682	43.690	2.796.202
4-way Mergesort	8.192	32.768	1.024	65.536	4.194.304
Funnelsort	8.192	32.768	1.024	65.536	4.194.304

Figure 7.1: The largest numbers of elements that fit within the various memory layers with a 4-byte element type when 32 MB of main memory is available.

interesting to see how execution time increases when the dataset approaches a size where it no longer fits in a certain memory layer, and cache misses therefore begin to occur at that layer. For those datasets the battle among the elements of who gets to stay in the cache between runs and who has to be thrown out to make room for others is intensified.

In order to measure the effects of the programs running out of main memory, quite large datasets are needed. As mentioned, since large datasets cause many page faults and make the programs run for a very long time, it would be nice to decrease the amount of memory available to the programs in order to reach the main memory boundary sooner. Fortunately, Linux can be booted with only a small amount of memory available and we have managed to boot the Pentium® with only 32 MB of main memory (recall, that the Athlon is used for dataset sizes smaller than main memory).

The size of all elements and pointers used is always 4 bytes, which seems appropriate on 32-bit computers. So for the 32 MB main memory boundary, the cache boundaries of both benchmarking computers and the main memory boundary of the Pentium are shown in Figure 7.1. The boundaries are shown in the number of elements the various memory layers can contain. They are calculated on basis of how much space the various data structures take up holding the indicated number of elements. Other data necessary for running the benchmarks are not included, so the numbers just indicate in which areas cache misses and page faults can be expected.

Aligning Data on Cache Line Boundaries

On page 58 we emphasized the importance of adjusting the size of the nodes in the heap layout to the cache line length in order to ensure optimal cache performance. This means that the addresses of the nodes must be divisible by 64, which is the cache-line length in bytes on the benchmarking computers. Only by aligning the tree at these boundaries can we ensure that reading a node into memory will incur only one cache block transfer.

An address is divisible by 64 when the 6 least significant bits of its binary representation are all unset, but when we allocate data we cannot be sure that this is the case. To allocate an aligned dataset we therefore allocate a chunk of memory that is a little larger than what we actually need, so that we can push the starting address a little without risking that the end of the dataset gets out of bound. If we allocate what corresponds to a cache line of extra data, then we can ensure that this will not happen.

If we want to allocate N bytes of data on a cache-line boundary, then we first allocate $N + 64$ bytes of data. Let the start address of these bytes be 0, and let P_{data} point 64 bytes into the data, that is, let P_{data} point to address 64. By manipulating P_{data}, so that

$$P_{data} = P_{data} \wedge \neg(64 - 1),$$

the pointer will now point to the first address between address 0 and 64 that is divisible by 64.

7.1.5 About the Results

All in all our benchmarks have resulted in ≈ 50 plots describing numerous characteristics of the different searching and sorting implementations. As it would be overkill to include and comment on all these plots, we have chosen a representative selection to illustrate the benchmarking results in the following two sections. The complete selection of plots can be downloaded from the web-site of this thesis.

Our main interest is in comparing the relative performance of the various programs. On many of the plots one program therefore forms a base line to which the other programs are compared.

7.2 Benchmarking the Static Search Trees Programs

In this section we present the results of benchmarking the static search trees. In general, all benchmarks are performed following the same strategy as Ladner et al. [30] who also carried out a series of experimental studies on static search trees. As a consequence, each benchmark case consists of N successful lookups, where N is the number of elements in the search tree. The sequence of elements to lookup is composed as a random permutation of the elements ranging from 0 to $N - 1$.

In Section 7.2.1 we carry out benchmarks to validate the analytically derived work complexities and branch misprediction counts. In Section 7.2.2 we investigate the way in which the various layouts use the memory system, and some interesting effects due to the set-associative cache of the Athlon is investigated in Section 7.2.4. Finally, we investigate the running times of all layouts for lookups of a random sequence of elements as well as another composition of the lookup sequence in Section 7.2.3.

7.2.1 Validating the Analytical Results

Before we get to validating the work complexities and branch misprediction counts, we investigate the quality of the pure-C programs. What we mean by quality in this context becomes clear in the following section.

Quality of Pure-C Programs

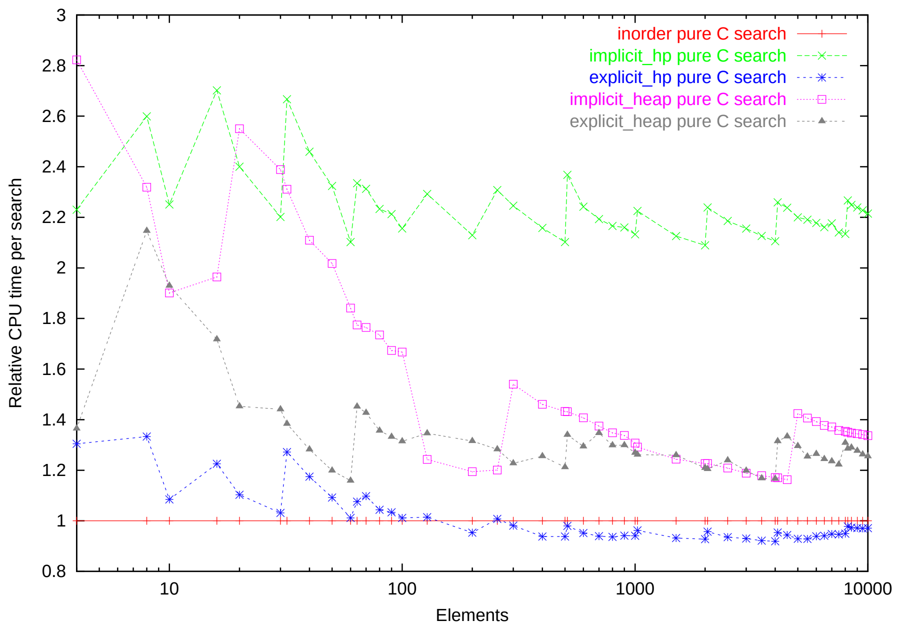
The pure-C search programs are translations of the generic search algorithm using different layout policies. This translation process is done by hand and resembles the work done by a compiler. It is therefore interesting to compare the behavior of the pure-C programs to that of the layout-based programs. Since the pure-C programs are C++ programs, we can compare their relative running times to those of the layout-base programs and thereby estimate their quality. Figure 7.2 shows the relative CPU usage of all search programs — pure-C versions and layout-based versions are shown in separate plots.

Overall, the compiler seems to make equally good executables of pure-C and layout-based programs, though there is a vague tendency of the pure-C programs being a little more efficient than the layout-based programs.

Work Complexity

We validate the analytically derived work complexities by comparing the relative number of instructions executed by the various layouts. Figure 7.3 shows the relative instruction counts of the layout policies on the Athlon computer. As predicted in Section 5.8 on page 79, the implicit height partitioned layout executes approximately two to three times as many instructions as the other layouts.

Furthermore, it is worth noting that the expected relative work complexities among the other four layouts quite precisely matches their relative instruction counts. The only mismatch is the instruction count for the in-order layout that seems to grow a little faster than the others. However, the overall picture indicates that the constant factors derived in the pure-C cost model can indeed be trusted as a measure of relative work complexity — even when the work complexities are as alike as is the case here.



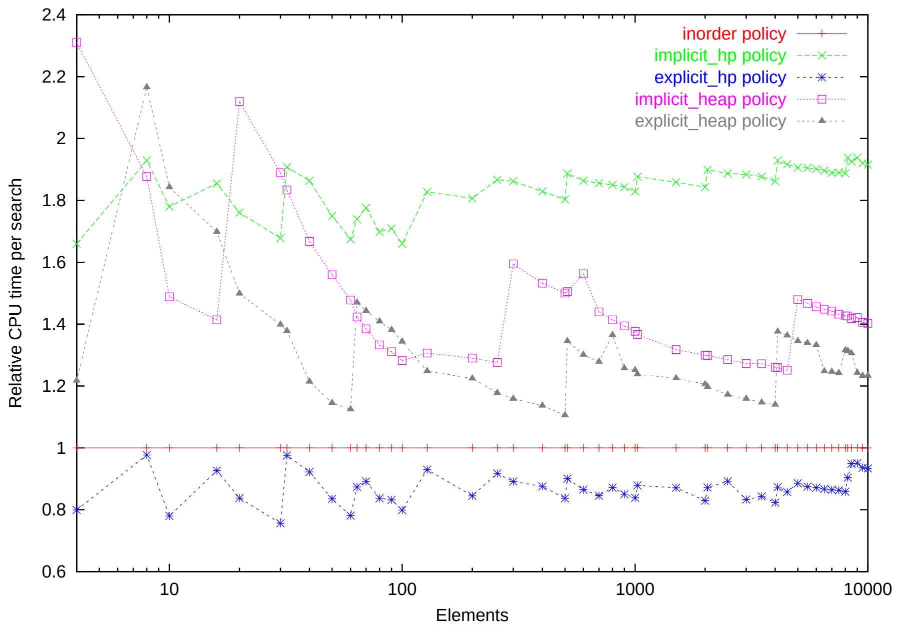


Figure 7.2: Relative CPU time measurements of the pure-C search programs (top) and the policy-based search programs (bottom) on the Athlon computer. The inorder method is used as base line.

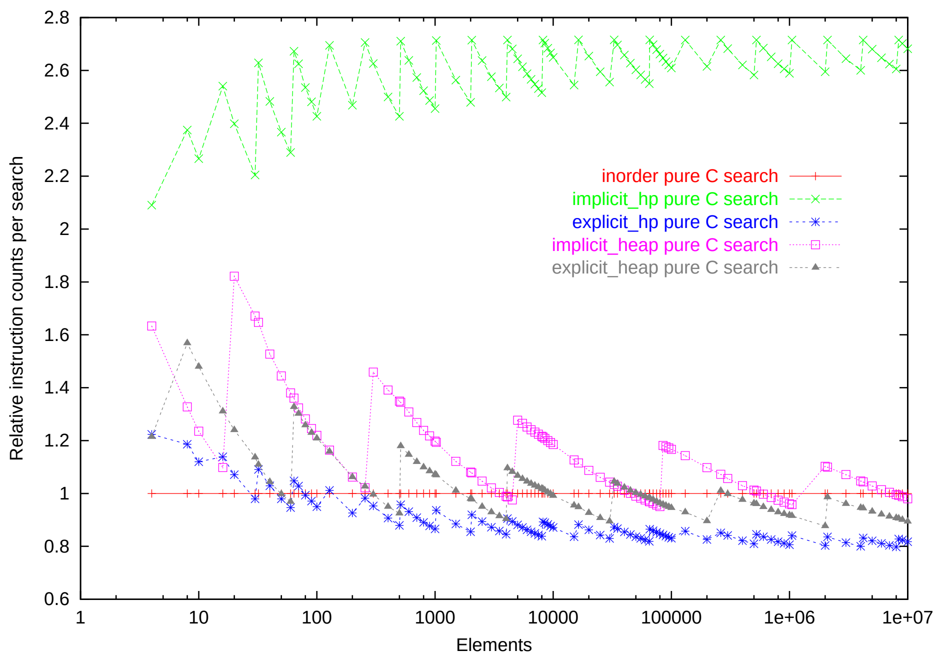
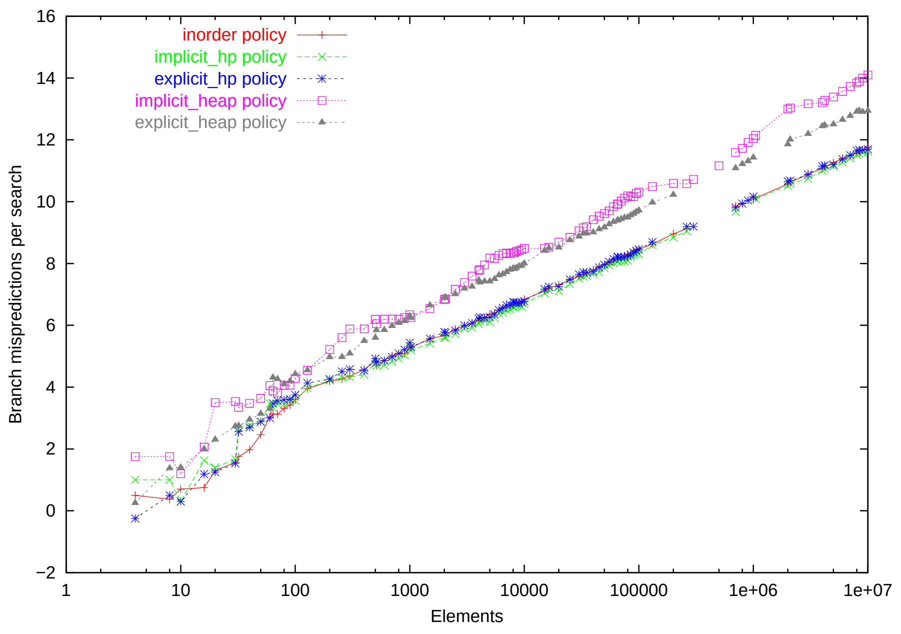


Figure 7.3: Relative instruction counts of the pure-C search programs on the Athlon computer with the inorder layout as base line.



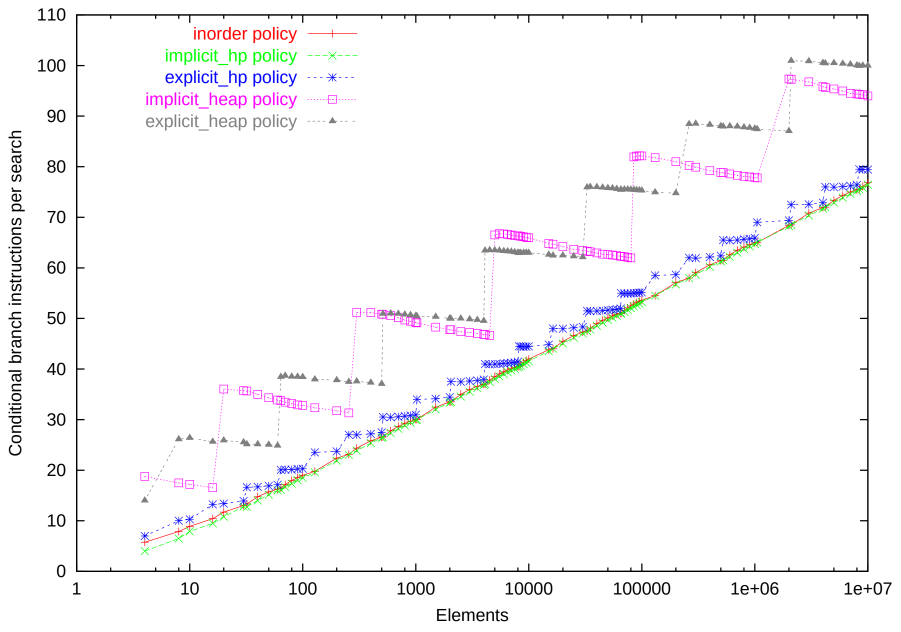


Figure 7.4: Branch misprediction counts (top) and conditional branch instruction counts (bottom) for the search tree layouts on the Athlon computer.

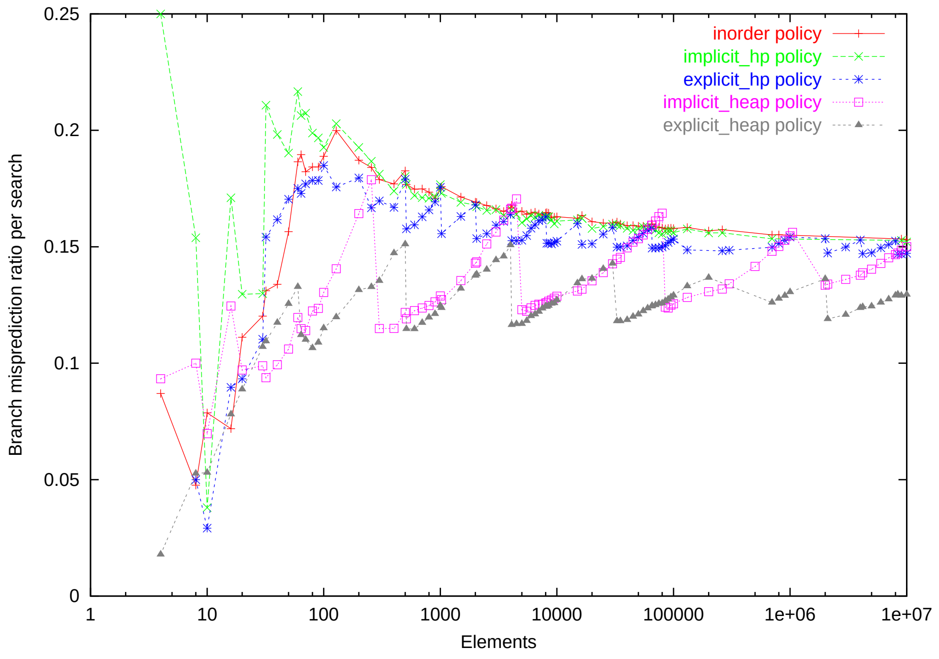


Figure 7.5: Branch misprediction ratios for the layout-based search trees.

Branch Mispredictions

Figure 7.4 (top) shows the branch misprediction counts of the various layouts. In the pure-C model we expected the explicit heap layout to incur the highest number of mispredictions but that is apparently not the case in practice. Though this layout incurs more mispredictions than the inorder and the height partitioned layouts, it actually incurs less mispredictions than the implicit heap layout. This is surprising since the implicit heap layout at the same time executes fewer conditional branch instructions than the explicit one (Figure 7.4, bottom). Looking at the branch misprediction ratios on Figure 7.5 we see that for some reason the Athlon is better at predicting the conditional branches of the explicit heap layout than those of the other layouts.

Recall, that in the pure-C model conditional branch instructions are divided into two categories based on whether they are easy or hard to predict. It seems that the pure-C model either needs a finer granularity of this categorization or a more precise prediction algorithm for the hard-to-predict branches than the one suggested by Mortensen [31]. Also, a better understanding of how compilers deal with conditional branches may lead to a higher precision in the pure-C model in this regard.

7.2.2 Investigating Memory Usage

In Section 5.8.2 on page 79 we argued that the method used for splitting the height partitioned tree into subtrees causes bad alignment of the subtrees for a cache-line length of 64-bytes. Also, we argued that the inorder layout is less affected by the cache-line length. It is therefore interesting to see if the theoretically superior cache complexity of the inorder layout pays off in practice.

Figure 7.6 shows the number of level 1 cache misses (top) and accesses (bottom) for the various layouts, while Figure 7.7 shows the number of level 2 cache misses[4].

The benchmarks do not support the expected superiority of the inorder layout. In fact, the implicit height partitioned layout exhibits better memory usage at both cache layers — even though it incurs a higher number of level 1 cache accesses than the inorder layout. The higher number of level 1 cache accesses can be explained by the precomputed table that the layout policy uses for navigation.

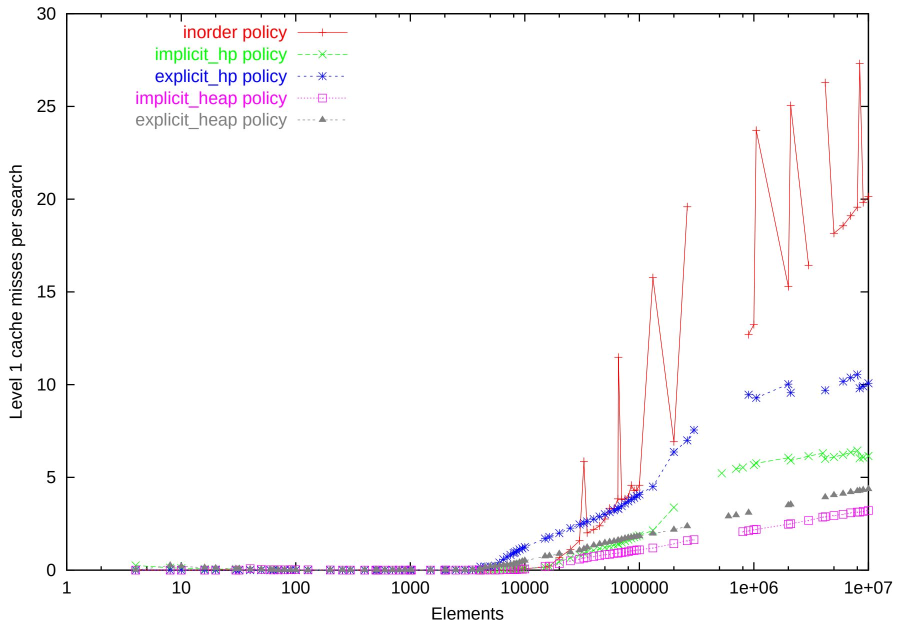
An explanation for the better memory usage of the implicit height partitioned layout can be found in a very recent draft paper of Bender et al. [5] that we became aware of just prior to the deadline of this thesis. In the draft, Bender et al. explore the cost of cache oblivious searching and argue that in the worst-case the number of cache misses incurred when following the path from the root to a leaf in the height partitioned tree (which is approximately $4 \log_B N$) only occurs for a small subset of the total set of such paths. Based on this observation, they argue that the *expected* number of cache misses incurred when following any path from the root to a leaf, is $2(1+1/\sqrt{B}) \log_B N$. Now, for these arguments Bender et al. use a splitting of the tree that results in equally large top and bottom recursive subtrees[5].

For uneven splits resulting in a top recursive subtree of height $\lceil ah \rceil$ and bottom recursive subtrees of height $\lfloor bh \rfloor$, $0 \leq a < b \leq 1$ and $a + b = 1$, they derive the optimal split to be when $1/4 \leq a < 1/2$. For these values of a they argue for an expected $(\log_2 e + \epsilon) \log_B N + O(1)$ cache misses, which is clearly better than for the even split.

However, our way of splitting the tree, so that the height of the bottom recursive subtrees are powers of 2, does not guarantee that a belongs to the range $1/4 \leq a < 1/2$. In fact, our way of splitting closely resembles the even split, since only top recursive subtrees can be split unevenly and all splittings of bottom recursive subtrees are even. Therefore, since our benchmark results are based on random lookups they should approximately follow the expected cache complexity derived by Bender et al. for the even split, that is, occurring $2(1+1/\sqrt{B}) \log_B N$ cache misses. Since this number

[4]Note that the number of level 2 cache accesses equals the number of level 1 cache misses.

[5]Figure 5.7 shows this splitting.



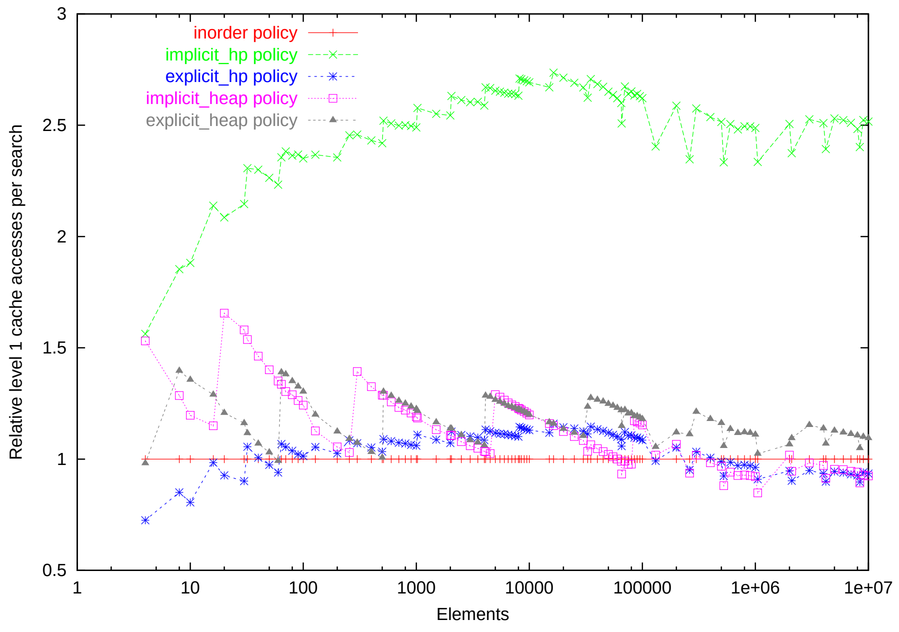


Figure 7.6: Level 1 cache misses (top) and relative level 1 cache accesses (bottom) for the layout-based search trees on the Athlon.

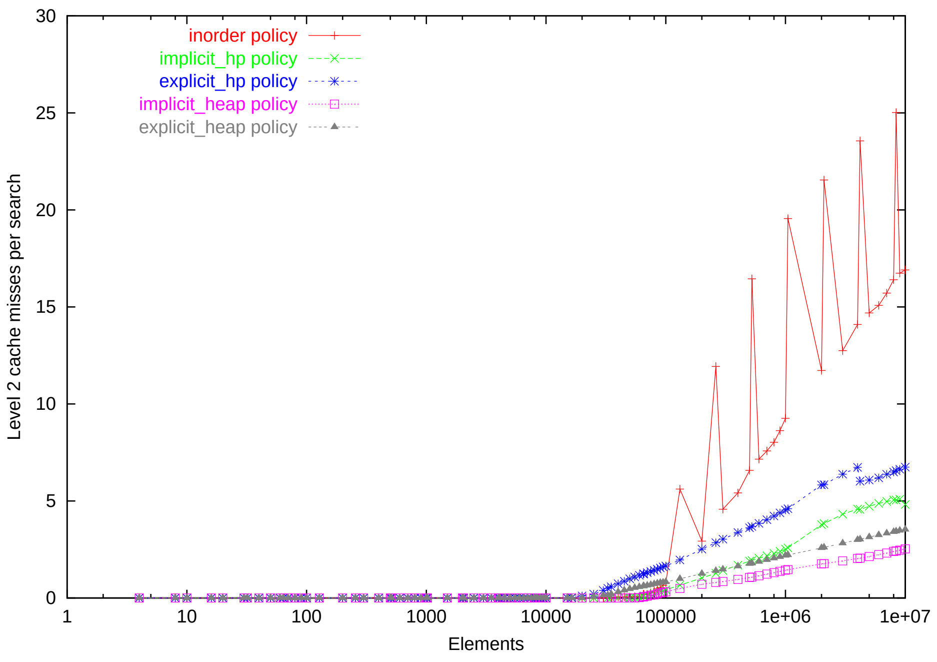


Figure 7.7: Level 2 cache miss counts for the layout-based search trees on the Athlon.

equals $5/8 \log_2 N$ for $B = 16$, the expected cache complexity of the implicit height partitioned layout grows slower than that of the inorder layout, which is $\log_2 N - 6$. This explains the better cache usage of the implicit height partitioned layout shown in Figure 7.6 and 7.7. It also explains why the explicit height partitioned layout is better than predicted.

If we ignore the height partitioned layouts, the relative expected cache efficiencies (see Section 5.11 on page 79) match the relative cache miss counts at both cache layers. The heap layouts are more efficient than the inorder layout, and the implicit heap layout is more efficient than the explicit heap layout.

From the plots we also see that the explicit layouts begin to cause more misses at both cache levels earlier than the other layouts. This is expected since they use extra memory for keeping child pointers. Overall, the dataset sizes for which the cache effects begin to stabilize matches the cache boundaries shown on Figure 7.1.

Due to the long running times when benchmarking large datasets, we have only sparse measurements of the main memory usage of the search tree layouts. Nevertheless, it is possible to draw a few conclusions from the page fault counts shown in Figure 7.8. We notice the same effect as with the cache miss counts that the explicit layouts incur page faults for smaller datasets than the other layouts. The curves of the explicit layouts also to grow faster than the others. The curves of the implicit layouts seem to grow slower than the curve of the inorder layout, but how much slower is difficult to determine.

7.2.3 Investigating Running Time

We have experienced how the complex navigational computations of the implicit height partitioned layout made it execute a significantly higher number of instructions than the other layouts. It is therefore interesting to see if the better memory usage of the implicit height partitioned layout makes it faster than especially the inorder layout.

Figure 7.9 shows the running times of the layout-based search trees measured in CPU time on the Athlon and in wall-clock time on the Pentium. The plots of the implicit layouts are the most smooth ones, which indicates that these layouts are least affected by the level 1 and 2 cache misses. On those plots the typical "knees bends", that mark the dataset sizes for which cache misses starts occurring at the next caching layer, are almost invisible. From being the least efficient layout for small datasets the implicit height partitioned layout becomes better than both the inorder layout and the explicit height partitioned layout when the datasets exceed the level 2 cache in size. It is hard to interpret exactly what happens when the datasets get larger than main memory, but the tendency is that the inorder layout ends with a higher growth rate than the implicit layouts, and that the implicit

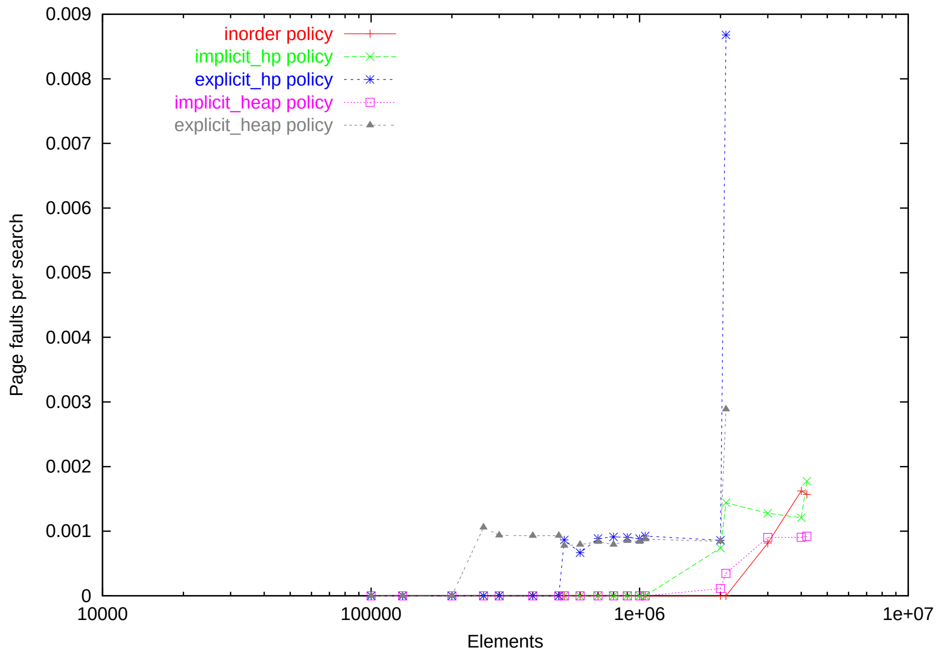
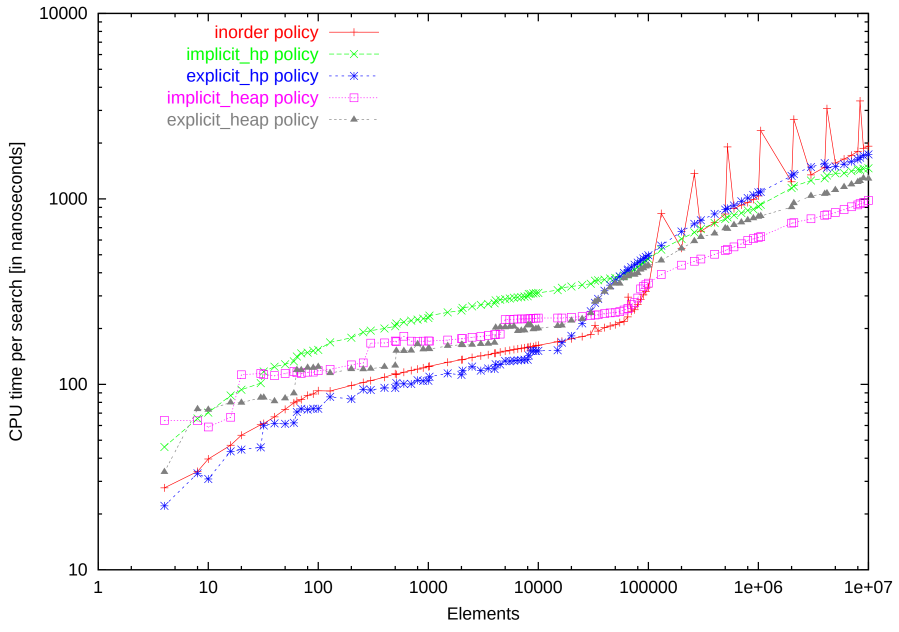


Figure 7.8: Page fault counts of the layout-based search trees on the Pentium.



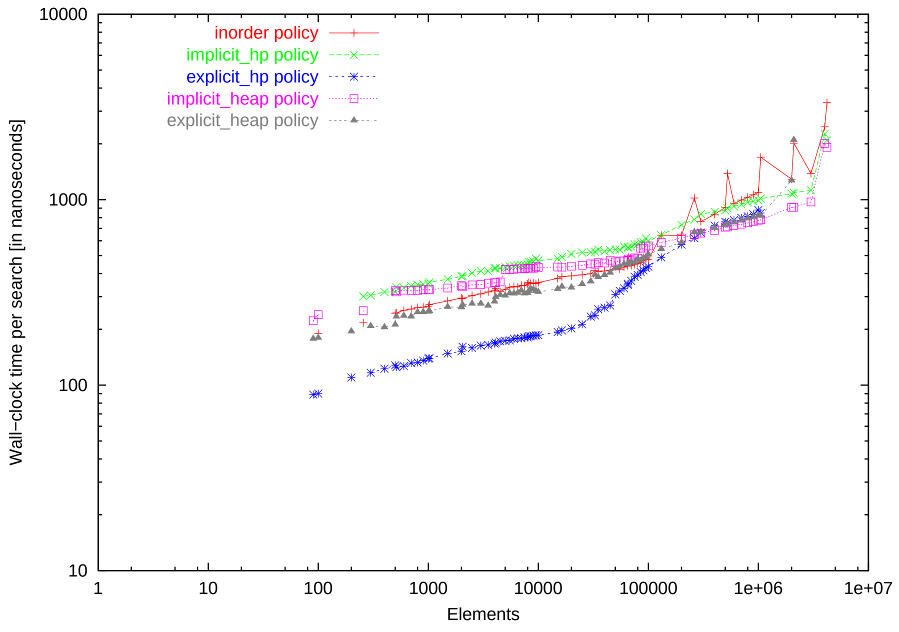


Figure 7.9: CPU time measurements on the Athlon (top) and wall-clock time measurements on the Pentium (bottom). The Pentium was booted with only 32 MB of main memory available in order to capture main memory effects.

height partitioned layout surely can compete with the explicit layouts.

It is interesting that the explicit height partitioned layout is the most efficient on both computers until the point where the level 2 cache misses begin to occur. It stays efficient on the Pentium until the dataset exceeds main memory where the high latency makes it unable to compete[6]. It seems that the better work complexity and branch misprediction count of the explicit height partitioned layout is worth more than the better level 1 cache usage of the explicit heap layout. The low level 1 cache miss penalty explains this relative efficiency.

This result contradicts the work of Ladner et al. [30] who found the cache-aware layout to be the most efficient. However, in the investigation by Ladner et al., the performance of the explicit cache-aware layout and the explicit cache-oblivious layout were very alike, so the reason for the conflicting results is most likely found in the implementational details.

7.2.4 Set-Associative Caches

The CPU time plot of the inorder layout (Figure 7.9, top) shows some sudden increases in the running time for a subset of the largest datasets sizes, i.e., for $2^{17}, 2^{18}, \ldots, 2^{23}$. As these tops are also visible on both the level 1 and 2 cache miss plots they are clearly related to memory usage.

The tops can be explained by investigating which memory addresses a search in the inorder layout accesses on the path from the root to a leaf, and how these addresses map to the cache-lines sets of the set-associative caches. If the dataset is of size 2^i (i.e., a power of 2), then each descend down the inorder tree will result in continuing the search in an array of size 2^{i-1}. Now, since the associativity of the caches are also powers of 2 the search keeps on accessing elements that map to the same set in the cache, and since these sets can only hold a limited number of blocks at a time[7], conflict cache misses begin to occur. Because the level 2 cache of the Athlon is 16-way set-associative these conflict misses occur when the tree has 16 or more levels, that is, when the dataset contains 2^{16} or more elements. On the contrary, when the dataset are large but not powers of 2 conflict misses are not an issue.

These cache effects are not captured by the ideal-cache model, as it assumes a full-associative cache.

[6]The missing measurements for the largest datasets on the Pentium (Figure 7.9, bottom) are due to the benchmarks running for too long, and not because of uncertainty in the measurements.

[7]On the Athlon these numbers are 2 and 16 for the level 1 and 2 caches respectively.

7.2.5 Composition of Lookup Sequence

In addition to benchmarking successful random lookups we have benchmarked the search trees for another composition of the lookup sequence. Figure 7.10 (top) shows that looking up elements of increasing value instead of random values completely removes the knee bends of the plots for all layouts. This is due to better memory usage. For all layouts, the sequence of memory blocks accessed during the search for the elements of value i and $i + 1$ are almost identical. Therefore, most of the blocks accessed when searching for $i + 1$ have been brought into the caches in the search for i, so fewer cache misses occur. The caches are said to be *warm*. By comparing the level 2 cache misses of the increasing lookup sequence (Figure 7.10, bottom) to that of the randomly permuted lookup sequence (Figure 7.7) it is apparent that the increasing sequence causes all layouts to incur fewer cache misses. The bad work complexity of the implicit height partitioned layout therefore becomes more influential on the running time, which is clearly shown in Figure 7.10 (top).

7.3 Benchmarking the Sorting Programs

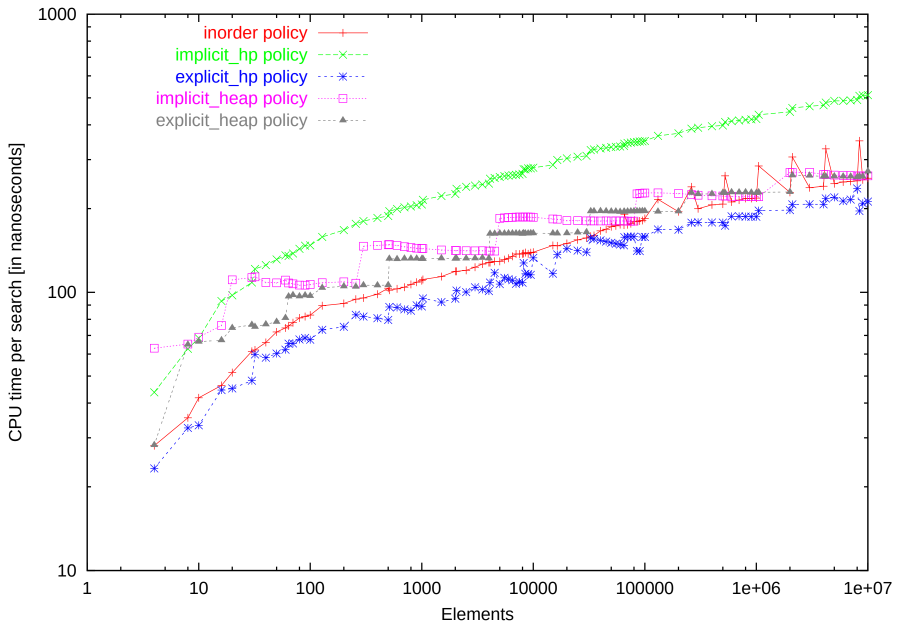
In this section we present the results of benchmarking the sorting algorithms. In each benchmark case a sequence of N elements is sorted. This sequence is a random permutation of the elements ranging from 0 to $N - 1$ and contains no duplicates. In order for us to interpret the results correctly, some comments on the funnelsort implementation we use should be made. We look into this matter in Section 7.3.1.

In addition to the 4-way mergesort and the funnelsort algorithms, we have chosen also to benchmark the introsort [33] algorithm available at SGI STL. Like the 4-way mergesort, this algorithm is also a state-of-the-art sorting algorithm. We have included a short description of introsort in Section 7.3.2.

In Section 7.3.3 we carry out benchmarks to validate the predicted work complexities. In Section 7.3.4 we look at the memory usage of the algorithms, and in Section 7.3.5 we investigate the running times of the algorithms.

7.3.1 The Funnelsort Implementation

Olsen & Skov [35] used funnelsort in their investigations of cache-oblivious priority queues. They compared the wall-clock time of heap construction when using funnelsort and introsort respectively. Their funnelsort implementation was the fastest for datasets exceeding the size of main memory, while introsort was superior for smaller datasets.



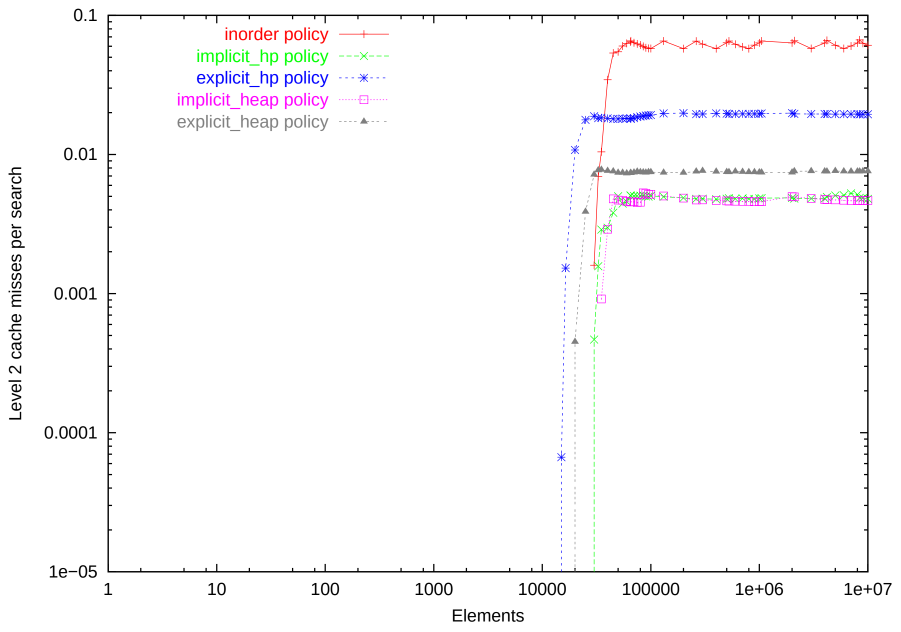


Figure 7.10: CPU time measurements on the Athlon for lookups of a sequence of elements of increasing value (top), and the number of level 2 cache misses that this sequence causes (bottom).

The implementation of Olsen & Skov is a combination of the funnelsort descriptions of Brodal & Fagerberg and Prokop [38], so it is not fully compliant to the description of Brodal & Fagerberg.

The implementation of Olsen & Skov uses cyclic buffers, which is not necessary for our purpose, since buffers are always filled completely and emptied completely. However, this will not influence the cache complexity of the algorithm. More importantly, the implementation does not use the idea described in Section 6.3 of merging the elements forth and back between two arrays. Instead, in every merge step, the elements are merged from the input array to an additional array, whereupon the elements are copied back to the input array. This incurs an additional $2\frac{\lceil N+1 \rceil}{B}$ cache misses for each level in the tree of funnels, so asymptotically this does not change the cache complexity of the algorithm.

Though the implementation of Olsen & Skov can be improved, the ways in which it differs from our description and analysis of funnelsort are minor. We will therefore use this implementation for our experiments[8]. The only change we have made to the implementation is introducing the parameter d.

7.3.2 The Introsort Competitor

The average running time of Quicksort is $O(N \log_2 N)$ when data are uniformly distributed. However, in the worst-case situation data are already sorted, which means that the quicksort partitions the elements in a maximally unbalanced way. As a consequence, the recursion becomes very deep, and the worst-case performance becomes $O(N^2)$.

Introsort uses quicksort as a primary sorting algorithm but can detect worst-case situations while the algorithm runs. The trick is to keep track of the recursion depth and switch to heapsort if the depth exceeds some predefined threshold. This way introsort works just as good as quicksort in the average case, and still exhibits $O(N \log_2 N)$ complexity in the worst case. As both quicksort and heapsort are in-place sorting algorithms, so is introsort.

7.3.3 Validating the Analytical Results

Figure 7.11 shows the relative instruction counts of 4-way mergesort, introsort, and funnelsort[9] using 4-way mergesort as a base line. In Section 6.5 we predicted that funnelsort would use approximately 8 times as many instructions as 4-way mergesort. This relation can be verified from the plot, which shows a factor between 6 and 10 for large datasets.

[8]The implementation can be downloaded from http://www.dunkel.dk/thesis/

[9]We have used $d = 3$ in the funnelsort benchmarks and introsort as base-case sorting algorithm unless otherwise noted.

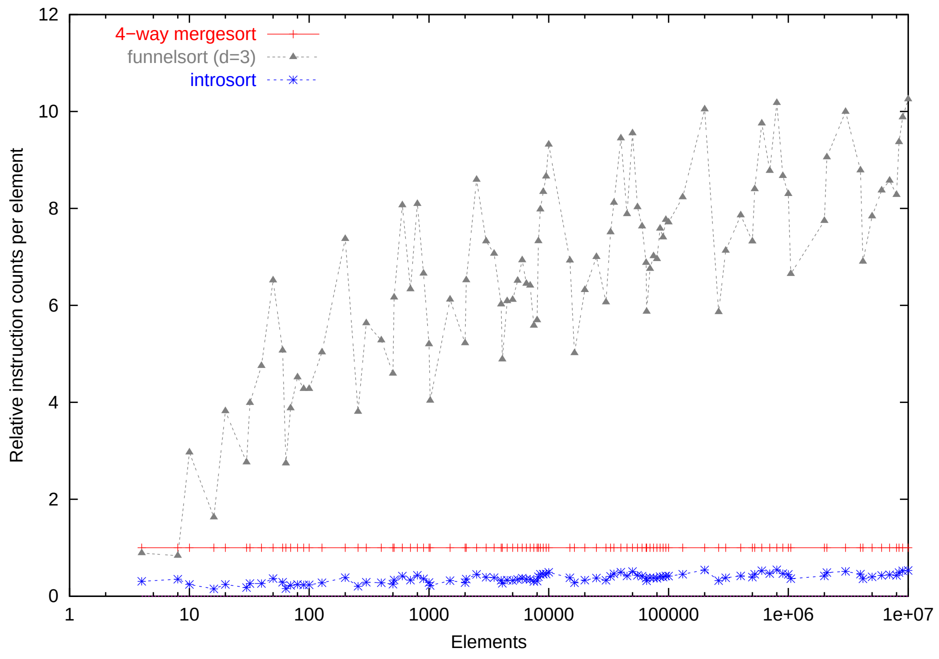


Figure 7.11: Relative instruction counts of the sorting programs on the Athlon computer with introsort as base line.

7.3.4 Investigating Memory Usage

The plots on Figure 7.12 do not match our cache complexity predictions of the 4-way mergesort being significantly superior to funnelsort. This verifies our suspicion in Section 6.5.1 that the derived worst-case cache complexity bound on funnelsort is not the tightest possible. As expected, the average-case performance is far better. The benchmarks indicate that the average-case cache complexity bound most likely can be found in the same area as the cache complexity bound of 4-way mergesort that we derived in Section 6.1.2.

Furthermore, the plots indicate that the cache complexities of 4-way mergesort and funnelsort are both superior to that of introsort. Introsort incurs fewer cache misses in the level 1 cache, but this is due to its better space complexity. The difference becomes apparent in the level 2 cache where the curve for introsort does not flatten out like the others. The page fault measurements for large datasets (Figure 7.13) show the same pattern as in the level 1 and 2 cache, that the two space consuming sorting algorithms reach the boundary between memory layers for smaller datasets than introsort. We would expect that the inefficient level 2 cache usage of introsort would also be evident from the page fault counts. However, on Figure 7.13 it is not possible to determine which of the three curves that exhibits the highest growth rate.

7.3.5 Investigating Running Time

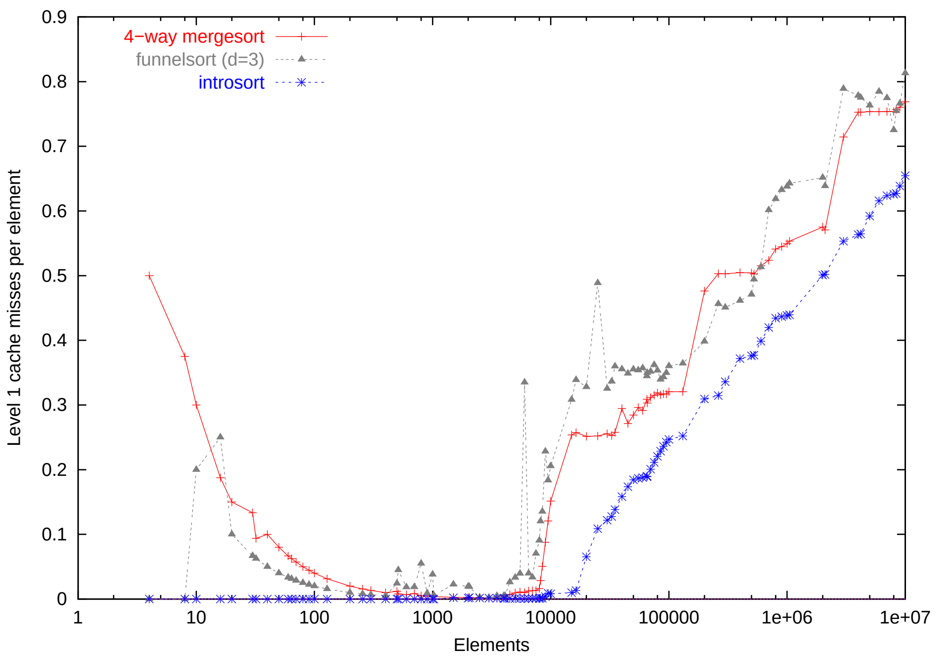
Figure 7.14 shows the CPU time on the Athlon and wall-clock time measurements on the Pentium of the sorting programs.

On the Athlon the 4-way mergesort and introsort seem to become equally efficient for large datasets. The introsort is superior to funnelsort for all input sizes in spite of its inferior cache performance.

On the Pentium, funnelsort seems to gain upon the 4-way mergesort for the largest datasets, while the introsort suffers heavily from reaching the upper boundary of main memory. The sparse measurements beyond main memory makes it impossible to predict how the relative performance will develop when the datasets becomes even larger.

7.3.6 The Parameter d

In Section 6.3.1 on page 94 we discussed the implications of tuning the parameter d. We argued that it might compromise the optimality of the cache complexity of funnelsort, since a large value of d would make the base-case sorting algorithm more influential. The level 2 cache miss counts on Figure 7.15 (top) verifies this prediction. The higher we choose d, the more cache misses the algorithm incurs. The plot on Figure 7.15 (bottom) shows how the different values of d influences the CPU time.



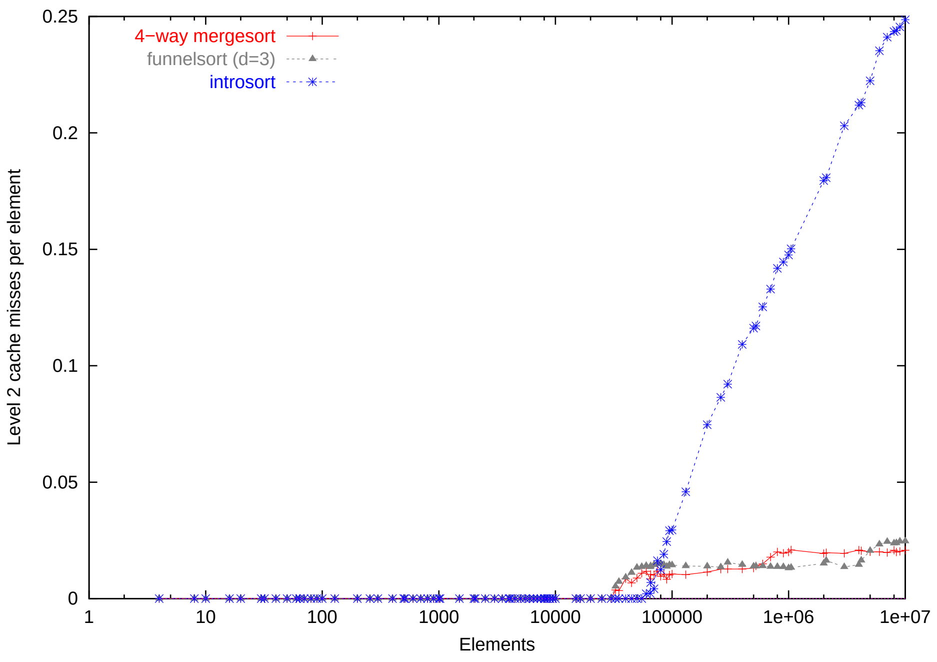


Figure 7.12: Level 1 cache misses (top) and level 2 cache misses (bottom) of the sorting programs on the Athlon computer.

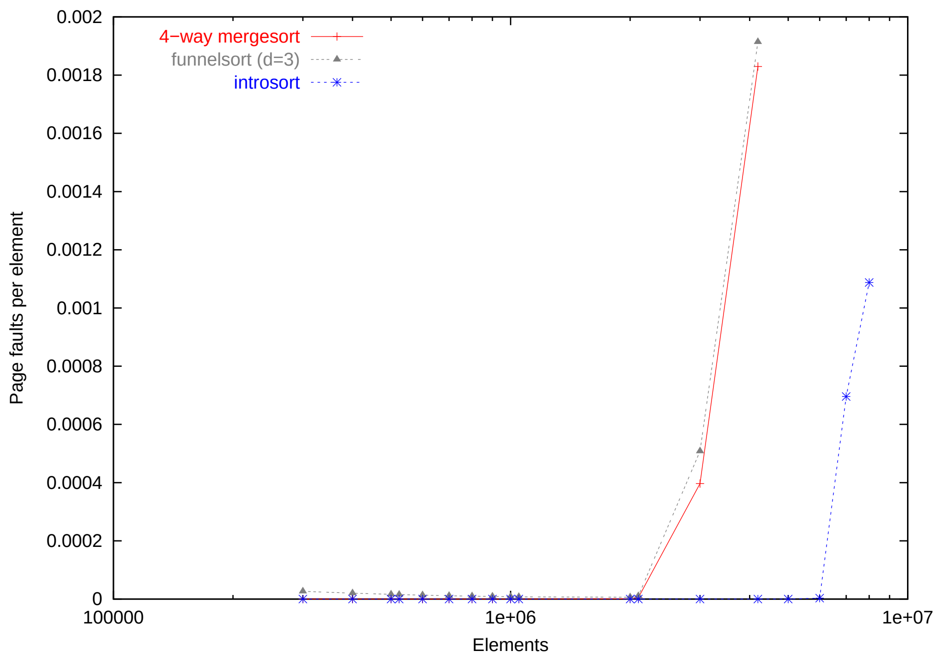


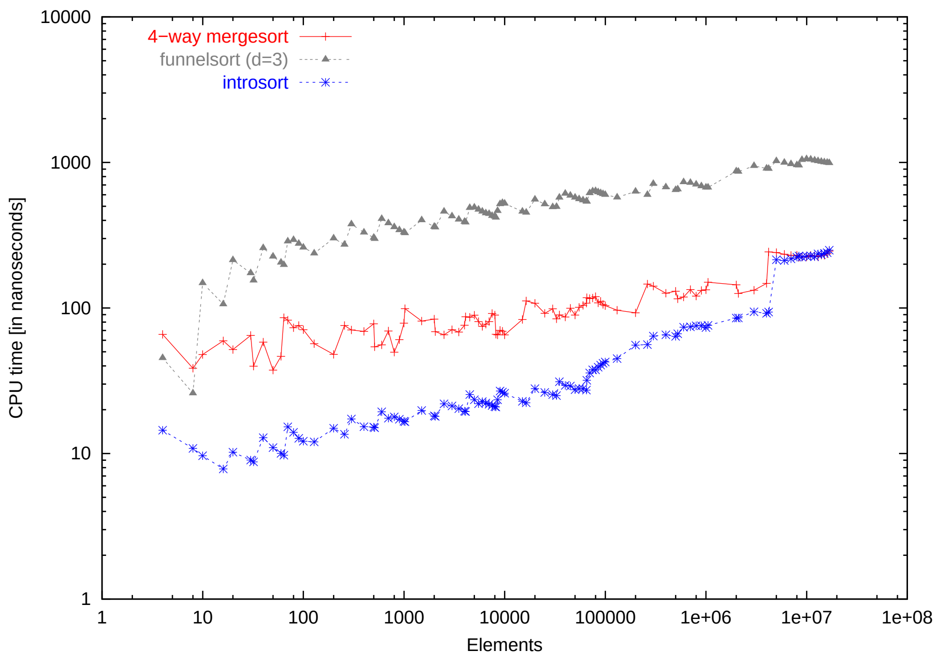
Figure 7.13: Page fault counts for the sorting programs.

7.4 Summary

In this chapter we have explained the methodology used for our benchmarks, including the way in which we measure memory usage, running time, and instruction counts. We have investigated the results of the searching and sorting benchmarks, and commented on these results separately.

Overall, the benchmark results have verified our predicted work and cache complexities of the search trees — though, the measured cache performance needed some further explanation on the average-case cache complexity of cache-oblivious search trees. The precision of the expected branch misprediction counts have not been equally good, which indicates that the pure-C model might need some adjusting in that respect.

The benchmarks of the sorting algorithms indicate that the average-case cache complexity bound on funnelsort is significantly better than the worst-case bound derived in Section 6.4.3. Our predictions of the work complexity of funnelsort relative to that of 4-way mergesort have proved successful.



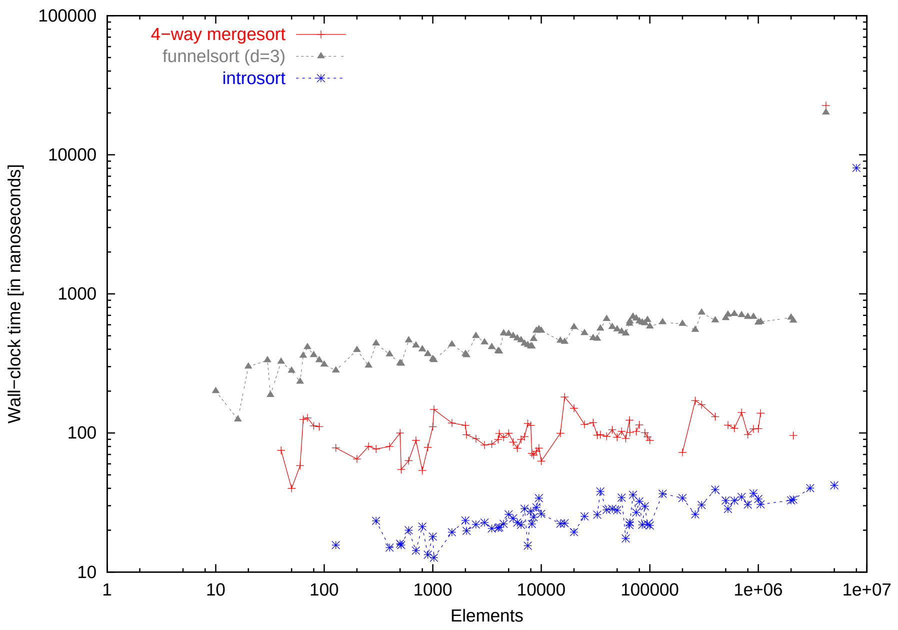
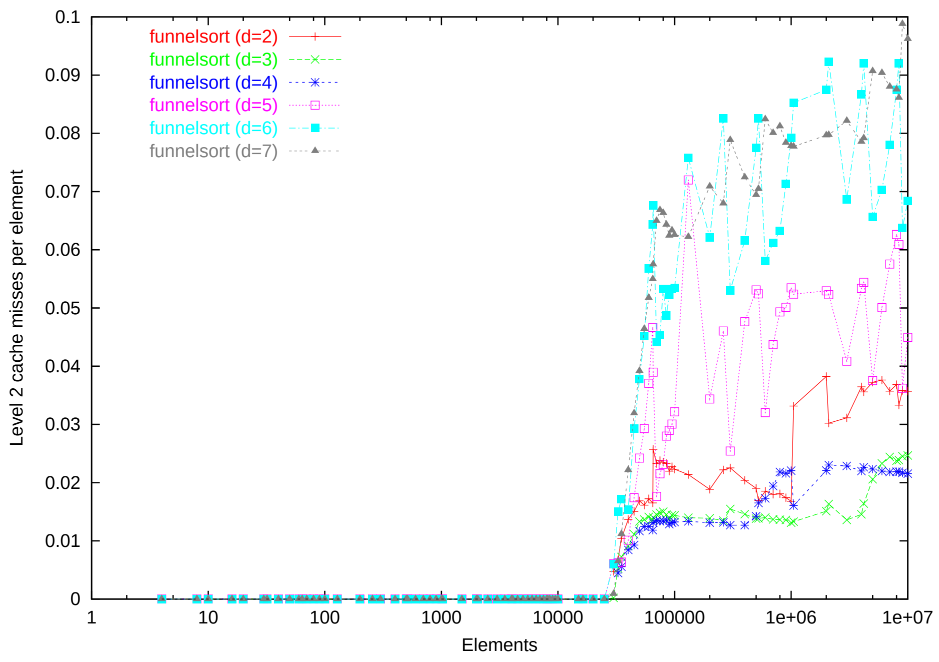


Figure 7.14: CPU time measurements on the Athlon computer (top) and wall-clock measurements on the Pentium computer (bottom) of 4-way mergesort, funnelsort, and introsort.



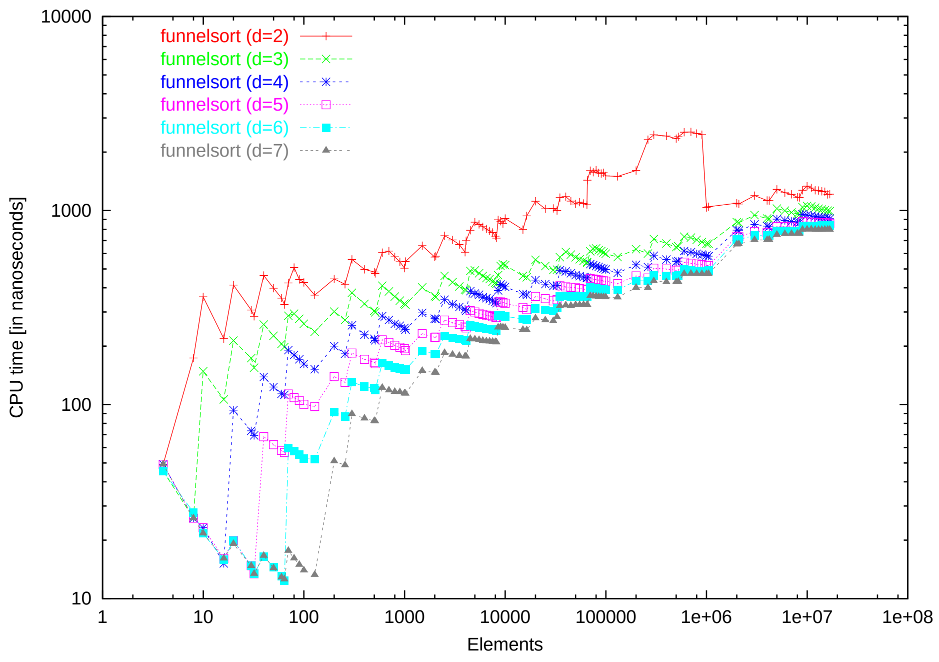


Figure 7.15: Level 2 cache miss counts (top) and CPU time measurements (bottom) of funnelsort for different values of d.

CHAPTER 8

Conclusion

"Ignorance is bliss."

— Franklin Pierce Adams

The main purpose of out work has been to investigate the possibilities of predicting the behavior of cache-oblivious searching and sorting algorithms by use of constant-factors analysis of work and cache complexity. Below, we summarize the main achievements in order of appearance:

- In Chapter 2 we looked into the way in which modern memory systems work, and saw how memory latency and the memory access pattern of an algorithm influences its running time. By use of an example we illustrated that the effects of efficient and inefficient memory access patterns are not captured in the RAM-model.

- The ideal-cache model and other models of memory systems capture the effects of memory access patterns. In the investigation of three such models in Chapter 3, we argued that the ideal-cache model offers some advantages to the other models, that makes it attractive. By investigating the theory behind the model we argued that it could be well-suited for estimating the relative memory efficiency of algorithms of similar asymptotic cache complexity.

- Earlier work on cache-oblivious algorithms indicates that such algorithms are more complex and incur higher instruction counts than traditional algorithms. Constant-factors analysis of cache complexity should therefore be companied by meticulous work complexity analysis. In Chapter 4 we argued that the pure-C model is the model best suited for such analysis due to its simplicity.

- In Chapter 5, we conducted a thorough analysis of the constant factors of the worst-case cache complexity of five search tree variants. Furthermore, we proved upper bounds on the pure-C instruction counts and branch misprediction counts of all five trees.

- The cache-oblivious search tree is of optimal cache-complexity due to a height partitioned memory layout. In Section 5.7.3 we have described a new algorithm that builds a tree in the height partitioned memory layout in linear time. This is an improvement of the, as far as we know, only other algorithm for this task described in the literature.

- In Chapter 6 we analyzed the 4-way mergesort and the cache oblivious funnelsort. We derived the constant factors of the worst-case cache complexity of both algorithms and of the cache-oblivious k-funnel data structure. Furthermore, we derived a bound on the pure-C instruction count of funnelsort.

- In Chapter 7 we ran thorough benchmarks, monitoring several aspects of the searching and sorting algorithms in order to evaluate the predicted performance.

  For the searching algorithms the benchmarks could only partially verify the predictions of relative work and cache complexities. However, with respect to the predictions of cache complexity, we have been able to explain the way in which the measured performance differ from the expected performance by use of a recent paper on average-case analysis of the cache oblivious search tree. When taking this analysis into account, our predictions matched the practical results closely. The same was true for the work complexity analysis, while the branch misprediction strategy of the pure-C model was unable to predict the relative branch misprediction counts precisely.

  The benchmarks of funnelsort indicate that our derived cache complexity bound indeed was a worst-case bound, and showed that funnelsort in the average-case behaved far better than predicted. In fact, the measured cache performance of funnelsort was comparable to that of the 4-way mergesort. This indicates that the average-case cache complexity bound of funnelsort most likely can be found in the same area as our derived worst-case cache complexity bound of the 4-way mergesort. Regarding the predictions of work complexity, the benchmarks verified the relative performance of funnelsort and the 4-way mergesort.

  The question whether the cache-oblivious approach is useful for searching and sorting algorithms is hard to answer in general. However, our results indicate that the use of pointer navigation was efficient in combination with the cache-oblivious search tree layout — at least for small input sizes. In fact, it proved even better than a cache-aware approach using pointers, but for larger datasets the cache-aware search trees were superior. Without use of pointers the traditional search tree was superior to the cache-oblivious search tree for small datasets, but the cache-oblivious search tree was superior for large datasets.

The cache-oblivious funnelsort was inferior to both introsort and 4-way mergesort in terms of running time for all dataset sizes. In spite of a superior cache complexity compared to that of introsort, the higher instruction count had more influence on the running time.

Working with the design and analysis of cache-oblivious algorithms has been both an inspiring and challenging process, which has left some impressions.

The ideal-cache model is neat. It makes cache complexity analysis manageable through the assumptions of a two-layered memory system of full associativity, with an optimal replacement strategy, and automatic data replacement. A strong property of the model is that knowing the details of the theory that validates these assumptions is no prerequisite for using the model. All that is needed is to know the implications of these assumptions.

The cache-oblivious approach gives an algorithm designer or analyst the freedom to think of algorithms in new ways. But with freedom follows responsibility. By the removal of the burden of considering memory system characteristics, the focus switches towards a more general way of designing memory efficient algorithms. The new techniques require new ways of thinking and thus may take time to master. However, the area of cache-oblivious algorithms is still new, so the most simple cache oblivious techniques and algorithms have probably not been discovered yet.

Therefore, cache-oblivious algorithms of better work complexity than the algorithms investigated in this thesis might prove the cache-oblivious approach even more useful than indicated by our results.

8.1 Directions of Further Work

There are a number of issues that we have chosen not to pursue in this thesis. Here follows a few suggestions for related further work:

- The benchmarking results of the inorder search tree reveal effects due to the set-associative cache of the Athlon computer. Certain dataset sizes incurred sudden increases in the memory performance of the inorder search tree. These cache effects are not captured by the ideal-cache model. In general, it would be interesting to investigate how cache-oblivious algorithms perform on direct-mapped or fully-associative caches. This might lead to guidelines on what kind of algorithmic behavior that should be avoided in order for an algorithm to be memory efficient.

- Our results indicate that the branch misprediction counts derived in the pure-C model in some cases do not match the actual behavior. Analyzing the branch misprediction units of modern CPUs might lead the way to a more realistic branch prediction strategy to use in the model. However, the strength of the pure-C model lies in its simplicity, so it is important that a new strategy is also simple.

- We have not paid much attention to optimizing the funnelsort algorithm in respect to work complexity, but our results suggest that there is room for improvement. Funnelsort needs scrutinizing in order to become competitive to other sorting methods. Most importantly, it should be possible to minimize the space used by the algorithm by modifying the merge procedure to work in-place. This technique might also be brought to work within a k-funnel itself, as it works with buffers in a similar way.

- Our results suggest that the average-case cache complexity of funnelsort is significantly better than our derived worst-case bound. The recent work of Bender et al. [5] on the average-case cache performance of the cache-oblivious search tree might provide for some inspiration on how such analysis could be conducted. An average-case analysis of funnelsort would lead to better understanding on the workings of the algorithms, which again could lead to a simpler and more efficient algorithm.

Bibliography

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, A Model for Hierarchiecal Memory, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, ACM Press (1987), 305–314.

[2] A. Aggarwal, A. K. Chandra, and M. Snir, Hierarchiecal Memory with Block Transfer, *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computing*, IEEE Computer Society Press (1987), 204–216.

[3] A. Aggarwal and J. S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Communications of the ACM* **31**,9 (1988), 1116–1127.

[4] 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.

[5] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz, The Cost of Cache-Oblivious Searching, Draft (2003).

[6] M. A. Bender, R. Cole, and R. Raman, Exponential Structures for Efficient Cache Oblivious Algorithms, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, Springer-Verlag (2002), 195–207.

[7] M. A. Bender, E. D. Demaine, and M. Farach-Colton, Cache-Oblivious B-Trees, *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press (2000), 399–409.

[8] 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.

[9] J. Bentley, Cache-Conscious Algorithms and Data Structures, World-wide Web Document (2000). Available at `http://www.cs.bell-labs.com/cm/cs/pearls/ccads.pps`.

[10] BitMover, LMbench, Worldwide Web Document (2003). Available at `http://www.bitmover.com/lmbench`.

[11] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: heap construction, *Journal of Experimental Algorithmics* **5** (2000), 15.

[12] G. S. Brodal and R. Fagerberg, Cache-Oblivious Distribution Sweeping, *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming, Lecture Note in Computer Science* **2380**, Springer-Verlag (2002), 426–438.

[13] 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*, Springer-Verlag (2002), 426–438.

[14] G. S. Brodal and R. Fagerberg, On the limits of cache-obliviousness, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing* (2003). To appear.

[15] G. S. Brodal, R. Fagerberg, and R. Jacob, Cache-Oblivious Search Trees via Binary Trees of Small Height, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM Press (2002), 39–48.

[16] S. A. Cook and R. A. Reckhow, Time-Bounded Random Access Machines, *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, ACM Press (1972), 73–80.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, *The MIT Electrical Engineering and Computer Science Series*, 4th Edition, MIT Press/McGraw Hill (2001).

[18] E. Demaine, *Cache-Oblivious Algorithms and Data Structures*, Preliminary lecture notes — handed out at the EFF Summer School on Massive Data Sets. June 27-July 1, BRICS, University of Aarhus.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, Cache-oblivious algorithms, *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press (1999), 285–297.

[20] T. Hagerup, Sorting and Searching in the Word RAM, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1373**, Springer-Verlag (1998), 266–298.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd Edition, Morgan Kaufmann Publishers Inc. (2003).

[22] International Organization for Standardization, *ISO/IEC 14882:1998: Programming Languages — C++*, International Organization for Standardization (1998).

[23] J. Katajainen and J. L. Träff, A meticulous analysis of mergesort programs, *Proceedings of the 3rd Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science*, Spring-Verlag (1997), 217–228.

[24] B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall (1988).

[25] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, 3rd Edition, Addison-Wesley (1998).

[26] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, 2nd Edition, Addison-Wesley (1998).

[27] D. E. Knuth, *The Art of Computer Programming, Fascicle 1: MMIX*, Addison-Wesley (1999). Available at `http://www-cs-staff.stanford.edu/~knuth/mmix.html`.

[28] D. E. Knuth, MMIX homepage, Worldwide Web Document (2003). Available at `http://www-cs-staff.stanford.edu/~knuth/mmix.html`.

[29] P. Kumar, Cache Oblivious Algorithms, *Algorithms for Memory Hierarchies*, *Lecture Notes in Computer Science* **2625**, Springer-Verlag (2003), 193–212.

[30] R. E. Ladner, R. Fortna, and B. H. Nguyen, A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation (2002).

[31] S. Mortensen, Refining the pure-C cost model, M. Sc. Thesis, Department of Computer Science, University of Copenhagen (2001).

[32] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press (1995).

[33] D. R. Musser, Introspective sorting and selection algorithms, *Software — Practive and Experience* **27**,8 (1997), 983–993.

[34] D. Ohashi, Cache Oblivious Data Structures, M. Sc. Thesis, University of Waterloo (2000).

[35] J. H. Olsen and S. C. Skov, Cache-Oblivious Algorithms in Practice, M. Sc. Thesis, Department of Computer Science, University of Copenhagen (2002).

[36] University of Tennessee, Performance application programming interface, Worldwide Web Document (2003). Available at `http://icl.cs.utk.edu/projects/papi/`.

[37] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2rd Edition, Morgan Kaufmann Publishers Inc. (1996).

[38] H. Prokop, Cache-Oblivious Algorithms, M. Sc. Thesis, Massachusetts Institute of Technology (1999).

[39] D. D. Sleator and R. E. Tarjan, Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM* **28**,2 (1985), 202–208.

[40] A. Srivastava and A. Eustace, Atom: a system for building customized program analysis tools, *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, ACM Press (1994), 196–205.

[41] J. S. Vitter, External Memory Algorithms and Data Structures: Dealing with MASSIVE Data, *ACM Computing Surveys* **33**,2 (2001), 209–271.

[42] J. S. Vitter, External Memory Algorithms: Computing on MASSIVE Data, Worldwide Web Document (2002). Available at `http://www.brics.dk/MassiveData02/slides/vitter-Basic.ps`.

APPENDIX A

Programs

A.1 Generic Search

```
#ifndef __GENERIC_SEARCH_CPP__
#define __GENERIC_SEARCH_CPP__

template <typename RandomIterator, typename T, typename LayoutPolicy>
bool generic_search(RandomIterator begin,
                    RandomIterator beyond,
                    LayoutPolicy policy,
                    const T& value) {
  policy.initialize(begin, beyond);
  while(policy.not_finished()) {
    if (policy.node_contains(value)) {
      return true;
    }
    else
      policy.descend_tree(value);
  }
  return false;
}

#endif //__GENERIC_SEARCH_CPP__
```

A.2 Inorder Search

```
#include <iterator>
#include <iostream>

#ifndef __INORDER_SEARCH_CPP__
#define __INORDER_SEARCH_CPP__

template<typename RandomIterator, typename Value>
class inorder_search_policy {
  public:
    inorder_search_policy() { }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_begin = begin;
      m_beyond = beyond;
      m_len = beyond-begin;
    }
    inline bool not_finished() { return (m_len > 0); }
    inline bool node_contains(const Value& value) {
      m_half = m_len >> 1;
      m_middle = m_begin;
      m_middle = m_middle + m_half;
      return *m_middle == value;
    }
    inline void descend_tree(const Value& value) {
      if (*m_middle < value) {
        m_begin = m_middle;
        ++m_begin;
        m_len = m_len - m_half - 1;
      }
      else
        m_len = m_half;
    }
  private:
    typename std::iterator_traits<RandomIterator>::difference_type m_half;
    typename std::iterator_traits<RandomIterator>::difference_type m_len;
    RandomIterator m_begin, m_beyond, m_middle;
};

#endif //__INORDER_SEARCH_CPP__
```

```
#ifndef __INORDER_SEARCH_PURE_C_CPP__
#define __INORDER_SEARCH_PURE_C_CPP__
#include <iterator>

namespace pure_c {
 template <typename RandomIterator, typename T>
 bool inorder_search(RandomIterator begin,
                     RandomIterator beyond,
                     const T& value) {
    typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
  initialize:
    diff_type half, len = beyond - begin;
    T middle_value;
    RandomIterator middle;

    goto not_finished;
  descend_right:
    begin = middle;
    begin = begin + 1;
    len = len - half;
    len = len - 1;
  not_finished:
    if (len == 0) goto return_false; /* hint: not taken */
  node_contains:
    half = len >> 1;
    middle = begin + half;
    middle_value = *middle;
    if (middle_value == value) goto return_true;
    if (middle_value < value) goto descend_right;
  descend_left:
    len = half;
    goto not_finished;

  return_false:
    return false;
  return_true:
    return true;
 }
}
#endif //__INORDER_SEARCH_PURE_C_CPP__
```

A.3 Optimized Lower Bound

```
#ifndef __LOWER_BOUND_OPTIMIZED_PURE_C_CPP__
#define __LOWER_BOUND_OPTIMIZED_PURE_C_CPP__
#include <iterator>

unsigned char log_table[256] = { 0xff, // <--- rogue value
                                 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
                                 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
                                 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
                                 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
                                 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
                                 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
                                 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
                                 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                                 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 };
// calculates floor( log2( n ) )
inline size_t floor_log2(unsigned long n) {
  long rv = 0;
  if (n & 0xffff0000) {
    rv += 16; n >>= 16;
  }
  if (n & 0xff00) {
    rv += 8; n >>= 8;
  }
  return rv + log_table[n];
}

template<typename RandomIterator, typename T>
RandomIterator lower_bound_optimized(RandomIterator begin,
                                     RandomIterator end,
                                     const T& val)
{
  typename std::iterator_traits<RandomIterator>::difference_type n = end - begin;
  if (n == 0) return end;
  ptrdiff_t i = (1 << floor_log2(n)) - 1;
  begin = begin[i] < val ? begin + (n - i) : begin;
  while (i > 0) {
    i = i >> 1;
    begin = begin[i] < val ? begin + i + 1: begin;
  }
  return begin;
}

#endif //__LOWER_BOUND_OPTIMIZED_PURE_C_CPP__
```

A.4 Implicit Heap Search

```
#ifndef __IMPLICIT_HEAP_POLICY_CPP__
#define __IMPLICIT_HEAP_POLICY_CPP__

#include <iostream>
#include <iterator>

using namespace std;

template <typename RandomIterator, typename Value>
class implicit_heap_policy {
  public:
    implicit_heap_policy(int degree) { m_degree = degree; }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_index = 0;
      m_size = beyond-begin-m_degree;
      m_current_node = begin;
      m_lower_bound = begin;
      m_begin = begin;
    }
    inline bool not_finished() { return m_index < m_size; }
    inline bool node_contains(const Value &value) {
      m_lower_bound =
        std::lower_bound(m_current_node, m_current_node+m_degree-1, value);
      return (*m_lower_bound == value);
    }
    inline void descend_tree(const Value& value) {
      m_index =
        m_index*m_degree + (m_degree-1)*((m_lower_bound-m_current_node)+1);
      m_current_node = m_begin+m_index;
    }
  private:
    int m_degree;
    typename iterator_traits<RandomIterator>::difference_type
      m_index, m_size;
    RandomIterator m_current_node, m_lower_bound, m_begin;
};

#endif // __IMPLICIT_HEAP_POLICY_CPP__
```

```
#ifndef __LOWER_BOUND_PURE_C__
#define __LOWER_BOUND_PURE_C__

#define LOWER_BOUND(lower_bound, x) \
   l_len = degree_minus_1; \
   goto l_not_finished; \
  l_right: \
   lower_bound = l_middle; \
   lower_bound = lower_bound + 1; \
   l_len = l_len - l_half; \
   l_len = l_len - 1; \
  l_not_finished: \
   if (l_len == 0) goto l_return; /* hint: not taken */ \
  l_found: \
   l_half = l_len >> 1; \
   l_middle = lower_bound + l_half; \
   l_middle_value = *l_middle; \
   if (l_middle_value < value) goto l_right; \
  l_left: \
   l_len = l_half; \
   goto l_not_finished; \
  l_return: \
   x = *lower_bound;

#endif //__LOWER_BOUND_PURE_C__
```

```
#ifndef __IMPLICIT_HEAP_SEARCH_PURE_C_CPP__
#define __IMPLICIT_HEAP_SEARCH_PURE_C_CPP__
#include <iterator>
#include <algorithm>
#include <vector>
#include "lower_bound_pure_c.h"

namespace pure_c {
 template <typename RandomIterator, typename T>
 bool implicit_heap_search(RandomIterator begin,
                           RandomIterator beyond,
                           int degree,
                           const T& value) {
   typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
  initialize:
   RandomIterator lower_bound, l_middle, current_node = begin;
   diff_type l_len, l_half, index = 0, size = beyond-begin;
   T l_middle_value, x;
   int degree_minus_1 = degree - 1;

   goto not_finished;
  descend_tree:
   index = index * degree;
   x = lower_bound-current_node;
   x = x + 1;
   x = degree_minus_1 * x;
   index  = index + x;
   current_node = begin + index;
  not_finished:
   if (index >= size) goto return_false; /* hint: not taken */
  node_contains:
   lower_bound = current_node;
   LOWER_BOUND(lower_bound, x)
   if (x == value) goto return_true;
   goto descend_tree;

  return_false:
   return false;
  return_true:
   return true;
 }
}

#endif //__IMPLICIT_HEAP_SEARCH_PURE_C_CPP__
```

A.5 Explicit Heap Search

```
#ifndef __EXPLICIT_HEAP_LAYOUT_CPP__
#define __EXPLICIT_HEAP_LAYOUT_CPP__

#include <iostream>
#include <iterator>
#include <algorithm>
#include <cmath>

using namespace std;

template <typename RandomIterator, typename Value>
class explicit_heap_policy {
  public:
    explicit_heap_policy(int degree) { m_degree = degree; }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_current_node = begin;
      m_beyond = beyond;
    }
    inline bool not_finished() {
      return (m_current_node < m_beyond);
    }
    inline bool node_contains(const Value &value){
      m_lower_bound =
        lower_bound(&(m_current_node->e[0]), &(m_current_node->e[m_degree-1]), value);
      return (*m_lower_bound == value);
    }
    inline void descend_tree(const Value& element) {
      m_current_node =
        m_current_node->p[m_lower_bound-reinterpret_cast<Value*>(m_current_node)];
    }
  private:
    int m_degree;
    RandomIterator m_current_node, m_beyond;
    Value *m_lower_bound;
};

#endif // __EXPLICIT_HEAP_LAYOUT_CPP__
```

```
#ifndef __EXPLICIT_HEAP_SEARCH_PURE_C_CPP__
#define __EXPLICIT_HEAP_SEARCH_PURE_C_CPP__
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
#include "lower_bound_pure_c.h"


namespace pure_c {
 template <typename RandomIterator, typename T>
 bool explicit_heap_search(RandomIterator begin,
                           RandomIterator beyond,
                           unsigned int degree_minus_1,
                           const T& value) {
   typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
  initialize:
   T x, l_middle_value, *lower_bound, *l_middle;
   diff_type y, l_len, l_half;
   RandomIterator current_node = begin;

   goto not_finished;
  descend_tree:
   y = lower_bound - reinterpret_cast<T*>(current_node);
   current_node = current_node->p[y];
  not_finished:
   if (current_node >= beyond) goto return_false; /* hint: not taken */
  node_contains:
   lower_bound = reinterpret_cast<T*>(current_node);
   LOWER_BOUND(lower_bound, x)
   if (x == value) goto return_true;
   goto descend_tree;

  return_false:
   return false;
  return_true:
   return true;
 }
}
#endif //__EXPLICIT_HEAP_SEARCH_PURE_C_CPP__
```

A.6 Implicit Height Partitioning Search

```
#ifndef __IMPLICIT_HEIGHT_PARTITIONING_POLICY_CPP__
#define __IMPLICIT_HEIGHT_PARTITIONING_POLICY_CPP__

#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include "hp_precomputed_table.h"

template <typename RandomIterator, typename Value>
class implicit_height_partitioning_policy {
  public:
    implicit_height_partitioning_policy(precomputed_table<Value> *precomp_table) {
      m_precomp_table = precomp_table;
    }
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_current_depth     = 0;
      m_begin             = begin;
      m_current_bfs_index = 1;
      m_size              = beyond-begin;
      m_precomp_table->initialise();
    }
    inline bool not_finished() { return m_current_bfs_index <= m_size; }
    inline bool node_contains(const Value& value) {
      return m_begin[m_precomp_table->Pos[m_current_depth]] == value;
    }
    inline void descend_tree(const Value& value) {
      if (m_begin[m_precomp_table->Pos[m_current_depth]] > value)
        m_current_bfs_index = m_current_bfs_index*2;
      else
        m_current_bfs_index = m_current_bfs_index*2+1;
      m_current_depth++;
      m_precomp_table->Pos[m_current_depth] =
        m_precomp_table->Pos[m_precomp_table->D[m_current_depth]]+
        m_precomp_table->T[m_current_depth]+
        (m_current_bfs_index & m_precomp_table->T[m_current_depth])*
        m_precomp_table->B[m_current_depth];
    }
  private:
    precomputed_table<Value> *m_precomp_table;
    typename iterator_traits<RandomIterator>::difference_type
      m_current_depth, m_current_bfs_index, m_size;
    RandomIterator m_begin;
};

#endif // __IMPLICIT_HEIGHT_PARTITIONING_POLICY_CPP__
```

```
#ifndef __IMPLICIT_HP_SEARCH_PURE_C_CPP__
#define __IMPLICIT_HP_SEARCH_PURE_C_CPP__
#include <iterator>
#include <vector>

namespace pure_c {
 template <typename RandomIterator, typename Tp>
 bool implicit_hp_search(RandomIterator begin,
                        RandomIterator beyond,
                        RandomIterator Pos,
                        RandomIterator T,
                        RandomIterator B,
                        RandomIterator D,
                        const Tp& value) {
   typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
  initialize:
   diff_type current_depth = 0, current_bfs_index = 1, size = beyond - begin;
   Tp x, y, z, w;
   RandomIterator p;
   *Pos = 0;

   goto not_finished;
  descend_right:
   current_bfs_index = current_bfs_index*2;
   current_bfs_index++;
   current_depth++;
   p = D + current_depth;
   x = *p;
   p = Pos + x;
   x = *p;
   p = T + current_depth;
   y = *p;
   z = current_bfs_index & y;
   p = B + current_depth;
   w = *p;
   z = z * w;
   w = x + y;
   w = w + z;
   p = Pos + current_depth;
   *p = w;
  not_finished:
    if (current_bfs_index > size) goto return_false; /* hint: not taken */
  node_contains:
   p = Pos+current_depth;
   x = *p;
   p = begin+x;
   x = *p;
   if (x == value) goto return_true;
   if (x < value) goto descend_right;
  descend_left:
   current_bfs_index = current_bfs_index*2;
   current_depth++;
   p = D + current_depth;
   x = *p;
```

```
    p = Pos + x;
    x = *p;
    p = T + current_depth;
    y = *p;
    z = current_bfs_index & y;
    p = B + current_depth;
    w = *p;
    z = z * w;
    w = x + y;
    w = w + z;
    p = Pos + current_depth;
    *p = w;
    goto not_finished;

   return_false:
    return false;
   return_true:
    return true;
 }
}
#endif //__IMPLICIT_HP_SEARCH_PURE_C_CPP__
```

A.7 Explicit Height Partitioning Search

```
#ifndef __EXPLICIT_HEIGHT_PARTITIONING_POLICY_CPP__
#define __EXPLICIT_HEIGHT_PARTITIONING_POLICY_CPP__

#include "hp_precomputed_table.h"
#include "../helper/log2.h"
#include <iterator>

template <typename RandomIterator, typename Value>
class explicit_height_partitioning_policy {
  public:
    explicit_height_partitioning_policy() {}
    inline void initialize(RandomIterator begin,
                           RandomIterator beyond) {
      m_current_element = begin;
      m_current_bfs_index = 1;
      m_size = beyond-begin;
    }
    inline bool not_finished() { return (m_current_bfs_index < m_size); }
    inline bool node_contains(const Value& value) { return m_current_element->e == value; }
    inline void descend_tree(const Value& value) {
      m_current_bfs_index = m_current_bfs_index*2;
      if(m_current_element->e > value)
        m_current_element = m_current_element->left_child;
      else {
        m_current_element = m_current_element->right_child;
        m_current_bfs_index++;
      }
    }
  private:
    RandomIterator m_current_element;
    typename iterator_traits<RandomIterator>::difference_type
      m_size, m_current_bfs_index;
};

#endif //__EXPLICIT_HEIGHT_PARTITIONING_POLICY_CPP__
```

```
#ifndef __EXPLICIT_HP_SEARCH_PURE_C_CPP__
#define __EXPLICIT_HP_SEARCH_PURE_C_CPP__
#include <iterator>
#include <vector>
#include <iostream>

namespace pure_c {
 template <typename RandomIterator, typename T>
 bool explicit_hp_search(RandomIterator begin,
                         RandomIterator beyond,
                         const T& value) {
  initialize:
   typedef typename std::iterator_traits<RandomIterator>::difference_type diff_type;
   RandomIterator current_element = begin;
   T x;
   diff_type size = beyond - begin, current_bfs_index = 1;

   goto not_finished;
  descend_right:
   current_element = current_element->left_child;
   current_bfs_index = current_bfs_index*2;
   current_bfs_index++;
  not_finished:
   if (current_bfs_index > size) goto return_false; /* hint: not taken */
  node_contains:
   x = current_element->e;
   if (x == value) goto return_true;
   if (x > value) goto descend_right;
  descend_left:
   current_element = current_element->right_child;
   current_bfs_index = current_bfs_index*2;
   goto not_finished;

  return_false:
   return false;
  return_true:
   return true;
 }
}

#endif //__EXPLICIT_HP_SEARCH_PURE_C_CPP__
```