Managing Memory Hierarchies

Jesper Bojesen

M. Sc. Thesis

Department of Computer Science
University of Copenhagen, Denmark

May 8, 2000

Abstract

This thesis is submitted in partial fulfillment of the Danish 'Kandidatgrad i datalogi' at the Department of Computer Science, University of Copenhagen and was written during the period August 1999 – April 2000 under the supervision of Jyrki Katajainen.

The interaction of programs and memory hierarchies is studied. One characteristic feature of a memory hierarchy is that it puts significance on the computation history: the execution time of an instruction in a program depends strongly on the path of execution leading to that instruction. The influence of such time dependent architectural features on the running time of programs has been growing steadily over the last few decades, to a point where it can no longer be abstracted away in the cost model.

The main part of this thesis is two performance engineering case studies, in which the goal is to tune the programs under consideration to achieve the best possible performance on modern hardware. While the final arbiter of program performance is an experiment, which is inherently architecture dependent, the process of program tuning is strongly backed by a theoretical analysis, which provides a better understanding of the measured behaviour, and quantifies to what extent the performance depends on the details of the architecture.

The remaining part of the thesis concentrates on the special problems that a set-associative cache creates for the performance of programs and in particular for the analysis of them. In this part, the main interest is in providing an analytical cache model, for selected program classes, which has the potential to direct automatic compiler controlled program optimisations. Hence, the requirements for the model are somewhat stricter than was used in the performance engineering part of the thesis. In this part results from the literature are surveyed but no programs are written and no elaboration on the results are done.

It should be mentioned the the performance engineering part of the thesis is a joint effort with my advisor Jyrki Katajainen (Chapter 2 and 3) and Maz Spork (Chapter 3). A preliminary version of Chapter 3 was presented at the 3rd Workshop on Algorithm Engineering 1999, after which it was invited to be included in the ACM Journal of Experimental Algorithmics special issue, devoted to the 3rd Workshop on Algorithm Engineering 1999. We do not yet know if it has been accepted.

Acknowledgements

Thanks must go to my advisor Jyrki Katajainen for always insisting on a clear definition of problems and concepts. In particular I must thank him for bringing Lemma 3.1 to my attention. — Without that many of the proofs would have been reduced to hot air and hand-waving. Also the work on Chapter 3 was supported in part by the Danish Natural Science Research Council under contracts 9701414 (project Experimental Algorithmics) and 9801749 (project Performance Engineering).

Contents

Chapter 1

General introduction

The working title of this project was a somewhat rhetorical 'architecture dependent programming'. The rationale behind that title was that, if programs are written and analysed completely within a specific cost model, the result is programs which are tuned for a non-existent architecture; namely, the architecture defined by the cost model. (Rhetorically, one may say that there is no such thing as architecture independent programming.) While it is not necessarily a problem that the computational model does not match the hardware exactly, the continued development in CPU and memory architecture design has reached a point where the most widely accepted family of cost models, the random-access machines, can no longer be relied upon to produce a valid description of program behaviour.

The main sources to deviations from the RAM models are the instruction-level parallelism and the memory hierarchy, which both have the property that the execution time of an instruction depends, sometimes strongly, on the history of computation leading to that instruction. However, since the constants introduced by the memory hierarchy are much larger than the amount of instruction-level parallelism supported by current CPUs, and since it seems that current compilers are much more successful at exploiting instruction-level parallelism automatically than at tuning programs for a memory hierarchy, the focus of this thesis is on analysing the effect of the memory hierarchy on program performance.

The literature on hierarchical memories falls in two major strands, characterised by their goal. One strand is primarily concerned with improving the performance of the memory system by tuning hardware parameters such as cache replacement strategy and cache size, while the subject of study for the second strand is the tuning of programs to allow a better utilisation of the memory hierarchy. However, many of the tools and techniques developed are equally relevant to both ends.

One technique for evaluating the cache behaviour of a program is to measure the number of cache misses incurred during a run of the program under investigation. This can be easily done and with zero overhead, by utilising the hardware monitor which is present in most of todays microprocessors. While this method is conceptually simple and very accurate, it only provides a single data point. Very little information is gained that will explain why the program performs in a certain way, and it is generally not possible to infer what would have happened if the cache configuration or the input had been different. This is not desirable since we are often interested in what could have happened, rather than what actually did happen. Nevertheless, because of the low overhead and simple use, a hardware performance monitor remains a valuable tool to provide an initial understanding, and as a reference for theoretical results.

Another related technique is to apply a cache simulator to a memory trace of the program. This approach immediately allows varying the cache configuration and in particular to try out non-existent configurations. Also, with a suitable simulator it becomes possible to identify exactly which statements in the source code are incurring the misses. Unfortunately a simulation requires a memory trace which for realistic problem sizes may easily be in the gigabyte range, and also the simulation may require a non-trivial amount of processing.

The third technique for the evaluation of a programs cache behaviour, which is the subject of this thesis, is through analysis of the source code form of programs, either manually or automatically. The hope is to obtain a much better understanding of the behaviour of a program, possibly leading to either faster program execution or more predictable program behaviour, whatever is most desired.

In Chapter 2 the problem of interchanging two array segments in place is studied. Three methods, implemented in the C++ standard template library, are analysed in a hierarchical memory environment, and two new programs are developed. One program is a simple variation of one of the Standard Templete Library (SGI STL) programs, while the other is a complicated adaption of another of the SGI STL programs. Interestingly, even though all programs are predicted to be about equally fast when measured in traditional means, the slowest program is more than 30 times slower than the fastest on real hardware. Moreover, the supposedly fastest of the programs is actually the slowest.

In Chapter 3 the problem of constructing an implicit binary heap is considered in detail. The starting point for the study is the original programs by Williams and Floyd, but also a recent method by Fadel et al. is considered. Each of the programs is modified to take the memory hierarchy into account. However, in this chapter, the cost model employed is specified in much more detail, and is followed rigorously. Our cost model, weighted pure C, is an variation of the word-RAM model, extended with a penalty when a cache miss is incurred. Since pure C is just a generalised assembly language with C syntax, this model gives rise to some interesting exercises in programming.

Both of the chapters 2 and 3 are context free, meaning that reading can be done in any order, and that no dependicies to the rest of the thesis are present. However, to ease reading, the terminology used in the two chapters are on purpose kept similar.

In Chapter 4 the subject is analytical cache models for set associative cache configuations. Chapter 2 and 3 assumed that the cache was fully associative, but this is hardly a realistic assumption. However in Chapter 4 we see that the assumption greatly simplifies the analysis. Some results from the literature are surveyed. A general analysis confirms that a direct-mapped cache may perform as either the optimal, or the worst possible, offline strategy depending on details in the input, and further determines how much changes in the cache architecture may effect the performance of a program. The general analysis is followed by two examples of analytical cache models, which allows predictions to be made for certain kinds of programs. The program classes studied are programs which access memory mostly sequentially, and certain loop-nested array references, that frequently occur in numerical programs.

Chapter 2

Interchanging two segments of an array in a hierarchical memory system

joint work with Jyrki Katajainen

Abstract. The problem of interchanging two segments of an array is considered. Three programs available in the Standard Templete Library (SGI STL) are analysed in a multi-level memory environment, and a new tuned adaptation of one of the SGI STL programs is developed. Even though the tuned version is suboptimal in traditional terms, it behaves better on a computer with a hierarchical memory system running about two times faster than the fastest SGI STL program.

2.1 Introduction

Assume that we are given an array **A** of size N, which can again be part of some bigger array. Let $\alpha = x_0x_1 \ldots x_{m-1}$ and $\beta = x_mx_{m+1} \ldots x_{m+n-1}$ be two adjacent **segments** of **A** such that $N = m+n$. In the **segment-interchanging** problem the task is to change the contents of **A** from $\alpha\beta = x_0x_1 \ldots x_{m+n-1}$ to $\beta\alpha = x_mx_{m+1} \ldots x_{m+n-1}x_0x_1 \ldots x_{m-1}$. Alternatively, the problem can be formulated as follows: given an array **A** of size N, rotate all the elements of **A** (simultaneously) n positions to the right $(\bmod N)$ [or equivalently m positions to the left $(\bmod N)$]. As an additional requirement, the interchange should be carried out as space-efficiently as possible. For the sake of simplicity, we assume that the array elements are integers, each of which can be stored in a single computer word.

In the literature the problem has been discussed at least under the names: block interchange [FS66, Knu98, MS87], block transposition [Knu97], section swap [Gri81, GM81], vector exchange [DD81], vector rotation [Ben86], segment exchange [Du88], and array rotation [She97]. In our choice **segment interchange**, the selection of the verb **interchange** was quite arbitrary, but we use the term **segment** to avoid the collision with a separate concept **block** that means an entity in which the data is transferred between different memory levels. By using the terminology of the Standard Template Library (STL) (see, e.g., [Str97]), the term **array** means any sequence of elements stored in consecutive locations in memory. Moreover, we shall use the term **segment swapping** to denote the special case of segment interchanging where the segments to be interchanged are of the same size.

The task how to carry out the interchange *in-place*, i. e., by using only a few words of extra space, is often given as an intellectual exercise in programming and algorithm courses. A fast in-place routine for segment interchanging is definitely a cornerstone in many in-place algorithms, for instance, for merging (see, e.g., [DD81, Du88, GKP99]) and sorting (see, e.g., [Knu98, Exercise 5.5-3]). According to Bentley [Ben86, Section 2.3] the problem arises in applications such as in the implementation of the APL language, which provides rotation as a primitive operation on arrays, and in the implementation of a text editor. Several segment-interchanging routines are also available in the current distribution of the Silicon Graphics Inc. implementation of the Standard Templete Library (SGI STL, version 3.2).

Basically, there exists four different approaches for solving the problem, even though in different sources the implementation details can vary a little bit. The four methods are: the copying method (see, e.g., [She97]), the reversal method (see, e.g., [Knu98, Exercise 5.5-3] or [GM81, MS87]), the successive-segment-swapping method [GM81] (see also [Gri81, Section 18.1]), and the dolphin method [FS66] (see also [Knu97, Exercise 1.3.3-34]) as it was named by Gries and Mills [GM81]. The first of these methods needs extra space for its operation, whereas the others operate in-place. A description of the methods can be found in Section 2.3. To evaluate the goodness of the methods, two measures have been used in earlier research papers: the number of **element moves** performed and the number of **element exchanges** performed. That is, the first measure tells how many assignments of the form $t := \mathbf{A}[i]$ or $\mathbf{A}[i] := \mathbf{A}[j]$ are executed, and the second measure how many parallel assignments of the form $\mathbf{A}[i], \mathbf{A}[j] := \mathbf{A}[j], \mathbf{A}[i]$ are executed. When comparing the results derived, it is normally assumed that the execution of an element exchange requires three sequential assignments.

Let $\gcd(m, n)$ denote the **greatest common divisor** of m and n. It is known that $N + \gcd(m, n)$ is a lower bound for the number of element moves performed by any segment-interchanging method [DD81, She97], and that this bound is reached by the dolphin method [DD81, She97]. It is also known that $N - \gcd(m, n)$ is a lower bound for the number of element exchanges performed [MS87], and that this bound is reached by the successive-segment-swapping method [MS87, She97]. In spite of these results it has been pointed out in many contexts (see, e.g., [Ben86, Exercise 2.6-4] or [Du88]) that the reversal method is the fastest in-place method in practice, even if it is not optimal with respect to neither of the above measures. According to our observations, there are two reasons for this: 1) the dolphin method relies heavily on random access which can be slow in current computers with a hierarchical memory, whereas the memory access pattern of both the reversal method and the successive-segment-swapping method is sequential; 2) due to its simplicity the reversal method seems to perform a bit less instructions than the successive-segment-swapping method.

Most high-performance computers have several processors and a hierarchical memory, each memory level with its own size, speed, and cost characteristics. The different memory levels existent in the vast majority of computers are from the fastest to the slowest: register file, on-chip cache, off-chip cache, main memory, and disk storage. The processors have only access to the lowest memory level, the register file, so the data has to be there before it can be processed. The basic idea is to keep the regions of recently accessed memory as close to the processors as possible. If the memory reference cannot be satisfied by one memory level, i. e., in the case of a **miss** or a **fault**, the data must be fetched from the next level. Not only one datum is fetched, but a **block** of data in the hope that the memory locations in vicinity of the current one will be accessed soon. Normally, a miss at some level means that the processors must wait until the requested data has arrived in the registers. Most computers have separate caches for data and program code, so the cost caused by instruction reads can be disregarded as long as programs are reasonably short. Therefore, the performance of a program is mainly determined by its memory-access pattern when manipulating data.

In this paper we study the performance of the segment-interchanging methods in a hierarchical memory system. Our main purpose is to tune the existing methods for the highest memory level that is too small to fit the whole input array. We call that particular memory level the **small cache**, and the memory level above that the **large cache**. Throughout this paper, we let B denote the the size of the blocks transferred between these two memory levels, measured in elements; and M the capacity of the small cache, measured also in elements. Even if many cache-based systems employ set associative caches (for the technical details the reader is referred to the book by Hennessy and Patterson [HP96]), we assume in our analysis that the small cache is **fully associative**, i. e., when a block is transferred from the large cache to the small cache, it can be placed in any of the M/B cache lines. In the case that the capacity of the small cache is exceeded, some of the blocks inside the cache should be replaced by the new block. We assume that this block replacement is accomplished by using the least-recently-used (LRU) strategy.

When analysing the goodness of the methods, we count both the number of **memory references** performed, as recommended by Knuth in his recent book [Knu93, pp. 464–465], and the number of **block transfers** performed between the small cache and the large cache, in a very similar manner as proposed by Aggarwal and Vitter [AV88]. That is, we count the number of memory reads and block reads, as well as the number of memory writes and block writes. Since in segment interchanging each element has to be read at least once and written at least once $2N$ is a trivial lower bound for the number of memory references performed. If all these memory references could be carried out blockwise, we get the lower bound $2N/B$ for the number of block transfers performed.

If the analysis would merely be based on the number of memory references performed, it is not difficult to show that the dolphin method is again optimal. However, it turns out that none of the known methods is optimal with respect to the number of block transfers performed. The best earlier in-place solutions require $4N/B+O(1)$ block transfers, whereas the copying method performs only $3N/B + O(1)$ block transfers. However, by a more careful programming the reversal method can be improved to require $3N/B + O(1)$ block transfers. This bound can be reached even if B is not known beforehand. The real challenge was how to get below the bound $3N/B + O(1)$. We were able to develop a method, a complicated adaptation of the dolphin method, for which the number of block transfers performed is only a few percent larger than the lower bound $2N/B$. To get this result, M must be given as part of the input. The method developed is not fully in-place, but it requires a buffer of size M/c, where $c > 1$ is a small positive constant. The efficiency of the method also depends on the cache-line size B.

Our experimental results show that, whatever the problem size is, the new method is about two times faster than the fastest program available in the SGI STL, and more than B times faster than the original dolphin method.

The rest of the paper is organized as follows. In in Section 2.2 we explain the notation we use to denote algorithms. In Section 2.3 we describe the three known methods for segment interchanging, as well as analyse their performance. In Section 2.4 two improved programs are presented. The first method, described in Section 2.4.1, is a simple modification of the three reversals method, while the second method, which is a complicated adaption of the dolphin method, is presented in Section 2.4.2 along with an analysis of correctness and performance. Finally the paper is completed with some experimental results in Section 2.5 to verify the theoretical results.

2.2 Program notation

To denote the sequence $0, 1, 2, \ldots, N-1$ we can write either $[0, \ldots, N-1]$ or $[0, \ldots, N)$. This differs from the similar set notation using curly brackets in that we put significance in the order of the elements. Otherwise it is similar to sets in that we do not allow (nor need) any value to occur more than once in a sequence, hence most of the usual operations on sets; e. g. set difference, test for membership, etc., carries over to sequences with the natural definitions. In particular if T is a injective function and s is a sequence, we can write $T(s)$ as a shorthand for the image sequence $[T(s_1), T(s_2), \ldots]$.

Furthermore, we use an associative array notation similar to what is used in the awk programming language. An associative array is composed of two sequences of equal length. One sequence specifies the addresses of elements, and one sequence specifies the values of same elements. So if $\mathbf{A}$ is an associative array, we can test if some element s is in the array with $s \in \mathbf{A}$, we can access the value with $\mathbf{A}[s]$, and we can assign a new value to an element by $\mathbf{A}[s] := x$. Because of the close connection between sets and associative arrays, we will often use common set terminology to describe operations on arrays. Using associative arrays we can extend Dijkstra's parallel assignment to handle assignment of array subsets. Let $\mathbf{A}$ be an associative array, and let $s = [s_1, s_2, \ldots]$ and $t = [t_1, t_2, \ldots]$ be sequences, not necessarily of equal length. We can then write $\mathbf{A}[s] := \mathbf{A}[t]$ as a shorthand for $\mathbf{A}[s_1], \mathbf{A}[s_2], \ldots := \mathbf{A}[t_1], \mathbf{A}[t_2], \ldots$. If s and t are not of equal length, the length of the shorter sequence determines how many elements are assigned. With this notation the rotation problem can be solved by the single assignment $\mathbf{A}[[0, \ldots, m)], \mathbf{A}[[m, \ldots, N)] := \mathbf{A}[[m, \ldots, N)], \mathbf{A}[[0, \ldots, m)]$. Our task is to develop a solution in which all parallel assignments are free from data dependencies, so that translation into a more conventional programming language is simple.

2.3 Known methods

In this section the known methods for segment interchanging are reviewed, and their behaviour in a hierarchical memory is analysed.

2.3.1 The copying method

In the **copying method** the shorter of the given segments is copied to an auxiliary array, the longer of the segments is moved to its destination, and finally the saved copy is moved from the auxiliary array to its destination. The main drawback of this method is that an extra array of size $\min\{m, n\}$ is required for its operation. To calculate the size of a segment in constant time, random-access iterators must be provided, but even if the sizes of the segments cannot be calculated, it is still possible to implement the method by using bidirectional iterators. We leave the details for an interested reader. It should be noted that the copying method is not implemented in the STL library. Instead, STL provides a `rotate_copy` function which, given an output sequence of size N, will perform a rotate and a copy to the output sequence in a single operation.

It is clear that each of the elements in the shorter segment has to be read and written twice, whereas each of the elements in the longer segment is only read and written once. Hence, the method requires $4\min\{m, n\} + 2\max\{m, n\}$ memory references, which is always bounded from below by $3N$. When a segment of size m is read sequentially, it might happen that we must visit $\lceil m/B \rceil + 1$ blocks. The same might happen in the case of writing. Thus, the three successive scans can cause at most $3N/B + O(1)$ block transfers. The worst case happens when all the blocks read into the cache during the first scan are removed from the cache before they are requested again by the second scan. Clearly, the bounds $3N$ for the number of memory references and $3N/B$ for the number of block transfers are reachable if $M \leq m = n$. It should be pointed out that, due to the sequential access pattern, a similar kind of analysis is valid for all the memory levels below the small cache as well.

As the experiments reported by Shene [She97] show, the in-place methods have difficulties in beating this simple method. This was the case even if Shene used a naive implementation, where the second segment was always moved to the auxiliary array. This means that in the worst case the number of memory references could be as large as $4N - O(1)$, and that of block transfers $4N/B - O(1)$.

2.3.2 The three reversals method

Let α^R denote the reversal of segment α. In the **reversal method** the interchange is accomplished with three successive reversals, first forming $\alpha^R\beta$, then $\alpha^R\beta^R$, and finally $(\alpha^R\beta^R)^R$, which is $\beta\alpha$. Since the reversal of a segment is easily accomplished in-place, segment interchanging can also be carried out in-place. As pointed out both in [Gri81, p. 302] and in [MS87], this method is part of the folklore and its origin is unknown. The three reversals method is implemented in the SGI STL using bidirectional iterators.

It is clear that each element takes part in two reversals, that is, the element is both read and written twice. This gives the bound $4N$ for the number of memory references. The array is scanned sequentially by the three reversal loops, so the number of block transfers is bounded by $4N/B + O(1)$. As earlier, a similar kind of analysis would be valid for all the memory levels below the small cache.

2.3.3 The dolphin method

Let m and n be the sizes of the two segments to be interchanged, and let $N = m + n$ be the size of the input array. Choose any i_0 from the set $\{0, 1, \ldots, \gcd(m,n))$. Now the element in $\mathbf{A}[i_0]$ is moved to a temporary area in order to create an empty slot, and the element whose final destination is $\mathbf{A}[i_0]$, i. e., the one stored in $\mathbf{A}[i_1]$ where $i_1 = (i_0 + n) \bmod N$, is moved there. This new empty slot is again filled with the element stored in $\mathbf{A}[i_2]$, where $i_2 = (i_1 + n) \bmod N$. This process is continued until i_j is equal to i_0, at which point the element from the temporary area is moved to the empty slot. The segment interchange is complete when all the indices in set $\{0, 1, \ldots, \gcd(m,n))$ have been handled. A variant of this method was described by Fletcher and Silver [FS66], and it appears also in [Knu97, Exercise 1.3.3-34]. We call it the **dolphin method**, as it was named by Ted Nelson [GM81]. Pseudo-code for the dolphin algorithm is given in Figure 2.1

```
Rotate(A[0, ..., m, ..., N − 1]) :
  for coset := 0 to gcd(m, n) − 1
    hole := coset
    src := (hole + m) mod N
    tmp := A[hole]
    while (src ≠ coset)
      A[hole] := A[src]
      hole, src := src, (hole + m) mod N
    A[hole] := tmp
```

Figure 2.1: The dolphin algorithm.

The correctness of this method is based on a group theoretic argument. The permutation corresponding to the segment interchange has $d = \gcd(m,n)$ disjoint cycles, each of length N/d. Furthermore, the index set of the elements that belong to one cycle is $\{i_0, i_0 + d, i_0 + 2d, \ldots, i_0 + N - d\}$, so each element in $\{0, 1, \ldots, \gcd(m,n))$ belongs to a separate cycle.

In this method each position in the input array is read and written exactly once. Hence, the method guarantees the optimal bound $2N$ for the number of memory references performed. Also it is easy to see that it will execute $N + \gcd(m,n)$ element assignments, which was shown in [DD81, She97] to be a lower bound on the number of assignments for any algorithm solving the rotation problem.

However, the cache behaviour of the dolphin method is poor due to the jumping memory access pattern. Since the elements of a cycle are evenly distributed across the array, it is clear that any input where $\gcd(m,n) \geq B$ will incur a cache miss on every memory reference if $N/\gcd(m,n)$ is larger than the cache. Also in the common case, when $\gcd(m,n) < B$ is the cache performance bad. Consider for example any input were $N/m > M \wedge m \geq B$. The first condition ensures that M cache lines will be used before the pattern wraps around at the end of the array, while the second condition ensures that only one element is accessed in each cache line in each round. So there exists inputs which will cause the dolphin algorithm to perform a block transfer for each of the $2N$ memory references. Indeed, theorem 2.1 in Section 2.4.2 below will make it clear that the dolphin algorithm will perform close to $2N$ cache block transfers for almost any input which does not fit the cache.

Two more observations should be made. First, if the element size is larger than the block size, e.g. $B \leq 1$, the only cache-induced loss in performance is caused by mis-alignment of elements in memory. The dolphin method can then be expected to perform similarly to the reversal method and the successive-segment-swapping method to be described next. Second, if $\gcd(m, n) > 1$, we can solve the equivalent problem of size $N/\gcd(m, n)$ by treating every segment of $\gcd(m, n)$ elements as a single element, effectively exchanging the inner and outer loop in Figure 2.1. So according to the first observation the dolphin method can be made cache efficient when $\gcd(m, n)$ is much larger than B. The first implementation of the dolphin algorithm in [FS66] applied this idea, but avoided the need for a buffer space by requiring element exchanges instead of element moves. Unfortunately, this idea is not of much help since in the common case $\gcd(m, n)$ is small.

2.3.4 The successive segment swapping method

Let α and β be the two segments to be interchanged, and assume that α is the shorter of the two. The input may be either $\alpha\beta$, in which case the return value should be $\beta\alpha$, or it may be $\beta\alpha$, in which case $\alpha\beta$ should be returned. Further, let β' be a prefix of β that is as long as α, and let $\beta = \beta'\beta''$. Now the equal-sized segments α and β' are swapped, and the reduced problem; - either $\alpha\beta''$ if α was the first segment, or $\beta''\beta'$ otherwise; - is solved recursively. The key observation is that two equal-sized segments can be interchanged in-place simply by exchanging the elements one by one. Also, the tail recursion can be easily removed so no recursion stack will be necessary. This approach for the segment-interchanging problem was proposed by Gries and Mills in a technical report [GM81], but it appears also in the book by Gries [Gri81, Section 18.1]. In the SGI STL there is an elegant implementation of the method using forward iterators only. Pseudo-code for the SGI STL implementation is given in Figure 2.2.

As earlier, let m and n denote the length of the segments interchanged. Furthermore, let s_i denote the size of the smaller segment in the ith segment swap, where i can be as high as $N-1$ if $\min\{m, n\} = 1$. Each segment swap, will bring s_i elements into their final destination. The $2s_i$ elements under consideration are all read and written once, i. e., the number of memory references performed is $4s_i$. However, the last segment swap does a bit better since it brings $2\gcd(m, n)$ elements into their final destination with $4\gcd(m, n)$ memory references. Thus, the total number of memory references performed is $4N - 4\gcd(m, n)$.

To analyse the number of block transfers, it is observed that memory accesses are strictly sequential, except for the change in flow caused by the assignment *right* := *mid* in Figure 2.2, line 10. This assignment will only be executed when the smaller segment of size s_i is to the right, and it can cause extra memory block transfers only when the rightmost segment occupies more than $M/2B$

```
Rotate(A[0, ..., m, ..., N − 1]) :
  left, right, mid := 0, m, m;
  loop
    A[left], A[right] := A[right], A[left]
    left, right := left + 1, right + 1
    if (left = mid)
      if (right = N)  break
      mid := right
    else
      if (right = N)  right := mid
```

Figure 2.2: The successive-segment-swapping algorithm.

blocks, i. e. when $s_i \geq M/2$. This can happen at most $2N/M$ times, causing $2(M/2B+1)$ extra block transfers in each round, for a total of $(2N/M)2(M/2B+1) = 2N/B + 4N/M$ block transfers beyond the obligatory 2N/B transfers. In total the number of block transfers performed by the successive segment swapping algorithm is bounded from above by $4N/B + 4N/M$, which is close to $4N/B$ for typical cache sizes.

2.4 Reengineered programs

In this section we present a straightforward improvement of the three reversals method which for some reason did not make it into the SGI STL, and a new method, based on the dolphin method, which repairs its bad cache behavior with only a minor extra cost in terms of memory references.

2.4.1 The three reversals method revisited

A simple optimization of the three revesals method is possible by fusing the three reversal loops into one. Recall that the sizes of two non-intersecting segments of array $\mathbf{A}$ being interchanged are m and n, respectively. The first iteration of the fused loop will handle the four elements in $\mathbf{A}[0]$, $\mathbf{A}[m-1]$, $\mathbf{A}[m]$, and $\mathbf{A}[N-1]$. The unoptimized method would execute three element exchanges: $\mathbf{A}[0], \mathbf{A}[m-1] := \mathbf{A}[m-1], \mathbf{A}[0]$; $\mathbf{A}[m], \mathbf{A}[N-1] := \mathbf{A}[N-1], \mathbf{A}[m]$; $\mathbf{A}[0], \mathbf{A}[N-1] := \mathbf{A}[N-1], \mathbf{A}[0]$. But because these exchanges are now brought together, they can be optimized to five sequential assignments: $t := \mathbf{A}[0]$; $\mathbf{A}[0] := \mathbf{A}[m]$; $\mathbf{A}[m] := \mathbf{A}[N-1]$; $\mathbf{A}[N-1] := \mathbf{A}[m-1]$; $\mathbf{A}[m-1] := t$. If $s = \min\{m, n\}$ and $S = \max\{m, n\}$, this process can be repeated $\lfloor s/2 \rfloor$ times after which a similar kind of cyclic permutation of three elements must be carried out $\lfloor S/2 \rfloor - \lfloor s/2 \rfloor$ times. The remaining $\lfloor N/2 \rfloor - \lfloor S/2 \rfloor$ iterations will do ordinary element exchanges. Pseudo-code for the tuned three reversals method is given in Figure 2.3.

This optimized method will bring $N/2$ elements directly to their final destination, so these elements are read and written only once. The other half of the elements are still to be read and written twice. Hence, the number of memory references reduces to $3N \pm O(1)$. Even the optimized method traverses the array sequentially, visiting half of the elements twice, so the number of block transfers is bounded by $3N/B + O(1)$. Clearly this bound is reachable if $M \leq \min\{m, n\}$.

```
Rotate(A[0, ..., m, ..., N − 1]) :
  i, j, k, l := 0, m − 1, m, N − 1
  while (i < j ∧ k < l)
    A[i], A[j], A[k], A[l] := A[k], A[i], A[l], A[j]
    i, j, k, l := i + 1, j − 1, k + 1, l − 1
  comment: Only one of the next two loops will be executed.
  while (k < l)
    A[i], A[k], A[l] := A[k], A[l], A[i]
    i, k, l := i + 1, k + 1, l − 1
  while (i < j)
    A[i], A[j], A[l] := A[l], A[i], A[j]
    i, j, l := i + 1, j − 1, l − 1
  while (i < l)
    A[i], A[l] := A[l], A[i]
    i, l := i + 1, l − 1
```

Figure 2.3: The tuned three reversals method

Both variants of the reversal method scan the memory sequentially so they have good spatial locality, but in their practical behaviour there are some differences. In the standard implementation two iterators move simultaneously through memory in opposite directions, so at least a 2-way set associative cache is needed to avoid interference. In the first part of the tuned version four sequential access iterators are maintained simultaneously which requires more registers a 4-way set associative cache to guarantee good performance.

2.4.2 The dolphin method revisited

In this section we describe the new segment-interchanging method which is an adaptation of the dolphin method. It behaves similarly to the copying method when available buffer space is larger than the smaller segment.

The basic idea is to move nearby elements simultanously without giving up the optimality of the dolphin method. That is, most elements are moved directly to their final destinations, but this is done in a mostly blockwise manner so that spatial locality is achieved.

As earlier, we let m and n denote the respective sizes of the segments to be interchanged, and assume for simplicity that $m > n$. Further, we let M' denote the size of the **buffer** used. The actual size depends on the size of the cache, M. Suppose now that two neighbouring elements stored in $\mathbf{A}[k]$ and $\mathbf{A}[k+1]$ have been moved to the buffer. The empty slots are to be filled with the elements from the positions $(k+m) \bmod N$ and $(k+1+m) \bmod N$, respectively. Unfortunately, we cannot be certain that these two elements are adjacent in memory. The problem that must be solved is: how to ensure that the chunk of empty slots which moves across the array remains contiguous, and how to keep track of the positions that have already been processed. Futhermore, this book-keeping must be done implicitly using only a constant amount of extra space.

Notation

First we will introduce some extra notation for use by the analysis.

Indices and Index ordering. For this discussion indices into the array **A** are regarded as elements in the Euclidean ring $\mathbb{Z}_N$ with modular arithmetic. For indices $\overline{a}, \overline{b} \in \mathbb{Z}_N$ we say that $\overline{a} < \overline{b}$ if and only if the representatives a, b in the range $0, \ldots, N-1$ are ordered $a < b$.

The greastest common divisor sequences. Consider the intermediate numbers generated by Euclid's algorithm for calculating the greatest common divisor, $\gcd(m, n)$. Let $r_1 = m$, $r_2 = n$, and $r_i = r_{i-2} \bmod r_{i-1}$ for $i > 2$. (Note: we still assume that $m > n$.) This defines a finite sequence, here called **the remainder sequence**, where the last non-zero element is $\gcd(m, n)$. **The quotient sequence** associated with the remainder sequence is defined by $q_i = \lfloor r_{i-2}/r_{i-1} \rfloor$ for $i > 2$. With these definitions we have that $r_{i-2} = q_i r_{i-1} + r_i$ for $i > 2$. We also define **the weighted Fibonacci sequence** associated with the remainder sequence and the quotient sequence by $L_1 = L_2 = 1$ and $L_i = q_i L_{i-1} + L_{i-2}$ for $i > 2$. Note that $N = L_i r_{i-1} + L_{i-1} r_i$ for any $i > 1$ where $r_{i-1} \neq 0$.

The index translation function, $T_m : \mathbb{Z}_N \to \mathbb{Z}_N$, or in shorthand, T, is defined by $T(a) = a + m$

The index sequence, I, is the sequence of indices generated by the index translation function, T, when applied repeatedly to 0, i. e., $I_0 = 0$ and $I_i = T(I_{i-1}) = T^{\circ i}(0)$. The index sequence is cyclic with a cycle-length $C = N/\gcd(m, n)$.

A pattern P is any subsequence of the index sequence, given by a first element, $\mathrm{first}(P) = I_p$, in the index sequence and a length. The length of P, denoted $|P|$, is the number of elements in P, i. e., the last element in P is $\mathrm{last}(P) = I_{p+|P|-1}$. We use the notation $\min P$ and $\max P$ to denote the smallest respectively the largest of the elements in a pattern P. Patterns are not allowed to be longer than C.

Compound patterns. Two patterns P, Q can be spliced together, forming the compound pattern $P \| Q$, if $|P| + |Q| < C$ and if the first element of Q is equal to the successor of the last element of P, i. e., $\mathrm{first}(Q) = T(\mathrm{last}(P)) = \mathrm{last}(P) + m$.

Pattern equivalence. For two patterns P, Q to be considered equivalent it is required that $|P| = |Q|$ and, assuming $\mathrm{first}(P) \geq \mathrm{first}(Q)$, that $\mathrm{first}(P) - \mathrm{first}(Q) \leq \min P$. Since $\max Q - \min Q = \max P - \min P$ when $|P| = |Q|$, we can also use $\mathrm{first}(P) - \mathrm{first}(Q) \leq -\max Q$ as criteria for equivalence. The important property of equivalent patterns is that equivalence allows simple calculations of maximum and minimum values. If a compound pattern $P \| Q$ is formed from equivalent patterns, we find that $\max P \| Q = \max P$ and $\min P \| Q = \min Q$ still assuming that $\mathrm{first}(P) \geq \mathrm{first}(Q)$.

Analysis of the load-once case

In this part of the analysis we will restrict ourself to algorithms that only loads elements into the buffer area once. Informally the load-once algorithms can be seen as specialisations of the schema in Figure 11, which uses awk-like associative arrays to implement the buffer and to keep track of the current position of the empty slots.

```
Rotate(A[0, ..., m, ..., N − 1]) :
  hole := some subset of [0, ..., N)
  buffer[hole] := A[hole]
  while (hole ≠ ∅)
    foreach (i ∈ hole)
      if (T(i) ∈ buffer)
        A[i] := buffer[T(i)]
      else
        A[i] := A[T(i)]
    hole, buffer := T(hole) − buffer, buffer − T(hole)
```

Figure 2.4: Outline of a load-once blockwise dolphin algorithm.

Clearly, for this approach to work we must load at least one element from each of the $\gcd(m,n)$ cycles into the buffer-area. We also notice that if T is applied to a contiguous sequence of indices $[i, \ldots, i+k]$ the image sequence will also be contiguous unless it contains both of the indices 0 and -1. Thus if we initially load elements from the vicinity of 0, i. e. either $[0, \ldots, M')$ or $[-M', \ldots, 0)$ we can be certain that the index set will remain contiguous.

Next we present a theorem which specifies the order in which the elements in the vicinity of 0 are visited when T is applied repeatedly to 0. This theorem will allow selection of an appropriate buffer size M' and provide foundation for an implementation of a load-once algorithm which does not utilise associative arrays.

Theorem 2.1 *Assume that $m > n$ and let P be a pattern with starting point* $\mathrm{first}(P) = I_1 = m$ *and length* $|P| = L_j$ *for some* $j > 1$ *where* $r_j \neq 0$*. Then; if j is even:* $\mathrm{last}(P) = \max P = -r_j$ *and* $\min P = r_{j-1}$*; and if j is odd:* $\max P = -r_{j-1}$ *and* $\mathrm{last}(P) = \min P = r_j$*. Furthermore, if $m < n$ swap the words 'even' and 'odd' in the above.*

Proof: Proof by induction after j. Assume that $m > n$.

Start: Consider the pattern P with starting point $\mathrm{first}(P) = m$ and length $|P| = L_2 = 1$. It is clear that $I_{|P|} = I_1 = \max P = \min P = m = -n = -r_2 = r_1$. We will also need to establish that the theorem is correct for $|P| = L_3 = q_3 + 1 = \lfloor m/n \rfloor + 1$. But in this case is $\max P = m$ and $\mathrm{last}(P) = \min P = r_3$ as desired.

Step: Assume for simplicity that j is even. Let P be a pattern with starting point $\mathrm{first}(P) = m$ and length $|P| = L_j$. By definition is $L_j = q_j L_{j-1} + L_{j-2}$ so we can divide P into sub-patterns $P_1, \ldots, P_{q_j}$ and Q, where $|P_1| = \ldots = |P_{q_j}| = L_{j-1}$ and $|Q| = L_{j-2}$ so that $P = P_1 \| \ldots \| P_{q_j} \| Q$. Since j-1 is odd we know from the induction hypothesis that $\mathrm{last}(P_1) = \min P_1 = r_{j-1}$ and $\max P_1 = -r_{j-2}$. Hence the first element in P_2 is $T(\mathrm{last}(P_1)) = m + r_{j-1}$. We now observe that $\mathrm{first}(P_2) - \mathrm{first}(P_1) = r_{j-1} < r_{j-2} = -\max P_1$ and therefore the patterns P_1 and P_2 are equivalent and we get that $\mathrm{last}(P_2) = \mathrm{last}(P_1) + r_{j-1} = 2r_{j-1}$. Likewise we see that the patterns $P_1, \ldots, P_{q_j}$ are equivalent and that $\mathrm{last}(P_{q_j}) = q_j r_{j-1}$. Therefore is $\mathrm{first}(Q) = T(\mathrm{last}(Pq_j)) = m + q_j r_{j-1}$

Now consider the pattern Q' with starting point $\mathrm{first}(Q') = m$ and length $|Q'| = |Q| = L_{j-2}$. We get from the induction hypothesis that $\mathrm{last}(Q') = \max Q' = -r_{j-2}$ and that $\min Q' = r_{j-3}$. We calculate $\mathrm{first}(Q) - \mathrm{first}(Q') = q_j r_{j-1} < r_{j-2} = -\mathrm{last}(Q')$ and see that Q and Q' are equivalent and hence $\mathrm{last}(Q) = \max Q = -r_{j-2} + q_j r_{j-1} = -r_j$ and $\min Q = r_{j-3} + q_j r_{j-1}$.

To sum up we have that $\min P = \min\{\min P_1, \min Q\} = r_{j-1}$ and $\max P = \max\{\min P_{q_j}, \min Q\} = -r_j = \mathrm{last}(Q) = \mathrm{last}(P)$.

The cases where j is odd and where $m < n$ are analogous and we will omit the proof of these. □

With help from theorem 2.1 and the identity $N = L_j r_{j+1} + L_{j+1} r_j$ it is not difficult to write a simple load-once algorithm. Such a program is presented in Figure 2.5.

```
1  Rotate(A[0,...,m,...,N-1]) :
2    assert: r_j ≠ 0 ∧ (even(j) ⇔ m < n)
3    hole := 0; buffer[0,...,r_j) := A[0,...,r_j)
4    for i := 1 to L_{j+1} - 1
5      assert: r_j ≤ T(hole) ≤ N - r_{j+1} - r_j
6      A[hole,...,hole + r_j) := A[T(hole),...,T(hole) + r_j)
7      hole := T(hole)
8    assert: T(hole) = N - r_{j+1}
9    A[hole + r_{j+1},...] := buffer[0,...,r_j - r_{j+1})
10   if (r_{j+1} ≠ 0)
11     for i := 1 to L_j
12       assert: r_j + r_{j+1} ≤ T(hole) + r_{j+1} ≤ N - r_{j-1}, except when i = 1
13       A[hole,...,hole + r_{j+1}) := A[T(hole),...,T(hole) + r_{j+1})
14       hole := T(hole)
15     assert: T(hole) + r_{j+1} = r_j
16     A[hole,...] := buffer[r_j - r_{j+1},...,r_j)
```

Figure 2.5: The load-once blockwise dolphin algorithm.

To prove the correctness of the program in Figure 2.5 we must justify that no position is processed twice, and that N positions are processed. The latter is seen directly by the identity $N = L_{j+1} r_j + L_j r_{j+1}$ while the former is a result of theorem 2.1. The assertions in lines 8 and 15 are direct results of theorem 2.1, and it is quite obvious that the overlap with $[0, \ldots, r_j)$ is handled correctly. What remains is to justify the correctness of the assertions in lines 5 and 12, which ensures that no other overlaps with $[0, \ldots, r_j)$ occurs. Consider the assertion in line 5. The elements to fill the hole are to be taken from the sequence of indices $[T(\mathit{hole}), \ldots, T(\mathit{hole}) + r_j)$. The assertion $r_j \le T(\mathit{hole})$ is a direct result of theorem 2.1 and so is valid, but the upper bound on $T(\mathit{hole})$ provided by theorem 2.1 is only $T(\mathit{hole}) < -r_{j+1}$. Anyway the bound can be strengthened to $T(\mathit{hole}) \le -r_{j+1} - r_j$ by the following argument: Suppose that $T(\mathit{hole}) + r_j > -r_{j+1}$ at iteration $0 < t < L_{j+1}$. Then we have $T^{\circ L_{j+1}}(0) = -r_{j+1} \in T^{\circ t}([0, \ldots, r_j))$. But T is a bijection and hence is $T^{\circ(L_{j+1}-t)}(0) \in [0, \ldots, r_j)$ or in other words $T(\mathit{hole}) < r_j$ in iteration $L_{j+1} - t > 0$ which is in contradiction with the already established bound $r_j \leq T(hole)$. The lower bound on $T(hole)$ in line 12 is established with a similar argument.

Unfortunately we cannot be sure that there exists an $0 < r_j \leq M$. This problem will be solved in the next section.

Analysis of the multiple load case

The problem with the program in Figure 2.5 is that, given a fixed buffer size M, we cannot be certain that there exists an $0 < r_j \leq M$; and even if such a value exists, it may be that the candidate r_j is too small to allow efficient buffering. In fact it could happen that the only candidate is 1, in which case the blockwise algorithm has the same memory access pattern as the original dolphin algorithm. To remedy this problem we must somehow allow the use of buffer sizes larger than r_j.

As an intermediate step to help the analysis we use the definition $L_{j+1} = q_{j+1}L_j + L_{j-1}$ to split the for-loop in Figure 2.5, line 4 into a pair of nested loops and an extra loop. The resulting loops are presented in Figure 2.6. To improve readability, the rather uninteresting assignments $\mathbf{A}[hole, \ldots] := \mathbf{A}[T(hole), \ldots]$ have been exchanged with the place-holder *SegmentMove*.

This intermediate program is obviously correct since it is a trivial modification of the program in Figure 2.5, but the interesting attribute of the intermediate program is the assertions in the lines 5 and 7 which with some consideration will allow reloading of the buffer in a controlled fashion.

Let us take a closer look at the loop in Figure 2.6 line 4. The assertion in line 5 tells us that if we start with a buffer size $M' > r_j$ we will have to unload $M' - r_j$ elements after $L_j - 1$ segment moves. We have room for $\lfloor M/r_j \rfloor$ segments of size r_j in the buffer simultaneously and in line 5 the buffer already holds r_j elements, so if we fill the buffer in the top of the loop with $\lfloor M/r_j \rfloor r_j - r_j$ extra elements, i. e. $M' = \lfloor M/r_j \rfloor r_j$, we must reduce the number of iterations from q_{j+1} to $\lfloor q_{j+1} r_j / M' \rfloor$ which leaves $q_{j+1} \mod (M'/r_j)$ iterations to be handled separately. An implementation of this idea is shown in Figure 2.7. For readability we use a separate buffer area for the extra loads, but it should be emphasised that there is never loaded more than M elements into the two buffers simultaneously, so the combined buffers still fit the small cache.

It should be clear that the assertions in Figure 2.7 are correct, but a few comments are required to understand how loading and unloading of the buffer-area is managed.- Consider the unload in line 10. The assertion in line 9 tells that we want to move elements from the positions addressed by $\{(i-1)M' + r_j, \ldots, iM' + r_j)$. We can see in line 6 that of these, the elements $\{(i-1)M' + r_j, \ldots, iM')$ are in the buffer and so must be unloaded rather than moved. After unloading, the index variable *hole* points $M' - r_j$ positions short of the hole, and must be corrected as is done in line 11.

We could stop here and use the algorithm we have developed so far, with but a few modification to handle the case when $r_j = 0$, but further opportunity for tuning exists in the last part of the algorithm. The remaining part of the program is composed of the three for-loops in Figure 2.7 line 14 (processing $(q_{j+1} r_j \mod M')L_j$ positions), Figure 2.6 line 8 (processing $L_{j-1} r_j$ positions) and Figure 2.5 line 11 (processing $L_j r_{j+1}$ positions). By filling the extra buffer with the $q_{j+1} r_j \mod M'$ elements at position $hole + r_j$, we can start the first two loops running simultaneously. The shorter of the two iterates L_{j-1} times,

1 $Rotate(\mathbf{A}[0,\dots,m,\dots,N-1])$:
2   **assert**: $r_j \neq 0 \wedge (\text{even}(j) \Leftrightarrow m < n)$
3   $hole := 0;\ buffer[0,\dots,r_j) := \mathbf{A}[0,\dots,r_j)$
4   **for** $i := 1$ **to** q_{j+1}
5   **assert**: $hole = (i-1)r_j$
6   **for** $k := 1$ **to** L_j **{** $SegmentMove;\ hole := T(hole)$ **}**
7   **assert**: $hole = q_{j+1}r_j = r_{j-1} - r_{j+1}$
8   **for** $i := 1$ **to** $L_{j-1} - 1$ **{** $SegmentMove;\ hole := T(hole)$ **}**
9   **assert**: $T(hole) = N - r_{j+1}$

Figure 2.6: First part of the intermediate load-once blockwise dolphin algorithm.

1 $Rotate(\mathbf{A}[0,\dots,m,\dots,N-1])$:
2   **assert**: $r_j \neq 0 \wedge (\text{even}(j) \Leftrightarrow m < n)$
3   $M' = \lfloor M/r_j \rfloor r_j$
4   $hole := 0;\ buffer[0,\dots,r_j) := \mathbf{A}[0,\dots,r_j)$
5   **for** $i := 1$ **to** $\lfloor q_{j+1}r_j/M' \rfloor$
6   **assert**: $hole = (i-1)M'$
7     $extra\_buffer[0,\dots,M'-r_j) := \mathbf{A}[hole + r_j,\dots]$
8   **for** $k := 1$ **to** $L_j - 1$ **{** $SegmentMove;\ hole := T(hole)$ **}**
9   **assert**: $T(hole) = (i-1)M' + r_j$
10     $\mathbf{A}[hole,\dots] := extra\_buffer[0,\dots,M'-r_j)$
11     $hole := hole + M' - r_j$
12     $SegmentMove;\ hole := T(hole)$
13   **assert**: $hole = \lfloor q_{j+1}r_j/M' \rfloor M'$
14   **for** $i := 1$ **to** $q_{j+1} \mod (M'/r_j)$
15   **assert**: $hole = \lfloor q_{j+1}r_j/M' \rfloor M' + (i-1)r_j$
16   **for** $i := 1$ **to** L_j **{** $SegmentMove;\ hole := T(hole)$ **}**
17   **assert**: $hole = q_{j+1}r_j = r_{j-1} - r_{j+1}$

Figure 2.7: Blockwise implementation of the loop in figure 2.6 line 4.

1 **assert**: $hole = \lfloor q_{j+1}r_j/M' \rfloor M'$
2 $extra\_buffer[0,\dots,q_{j+1}r_j \mod M') := \mathbf{A}[hole + r_j,\dots]$
3 **for** $i := 1$ **to** $L_{j-1} - 1$ **{** $SegmentMove;\ hole := T(hole)$ **}**
4 **assert**: $T(hole) = N - q_{j+1}r_j \mod M' - r_{j+1}$
5 $\mathbf{A}[hole + r_{j+1},\dots] := buffer[0,\dots,r_j - r_{j+1})$
6 **for** $i := 1$ **to** $Lj - L_{j-1}$ **{** $SegmentMove;\ hole := T(hole)$ **}**
7 **assert**: $T(hole) = \lfloor q_{j+1}r_j/M' \rfloor M' + r_j$
8 $\mathbf{A}[hole,\dots] := extra\_buffer[0,\dots,q_{j+1}r_j \mod M')$
9 $hole := hole + (q_{j+1} \mod M')r_j$
10 **if** $(r_{j+1} \neq 0)$
11   **for** $i := 1$ **to** L_{j-1} **{** $SegmentMove;\ hole := T(hole)$ **}**
12   **assert**: $T(hole) + r_{j+1} = r_j$
13   $\mathbf{A}[hole,\dots] := buffer[r_j - r_{j+1},\dots r_j)$

Figure 2.8: Combining the loops in Figure 2.7 line 14, Figure 2.6 line 8 and Figure 2.5 line 11.

so we must divide the loop in two; - One that iterates $L_{j-1}-1$ times and one that iterates $L_j - L_{j-1}$ times. The combined loops are listed in Figure 2.8.

$Rotate(\mathbf{A}[0,\ldots,m,\ldots,N-1])$:
  assert: $r_j \le M \wedge r_{j-1} \neq 0 \wedge (\text{even}(j) \Leftrightarrow m < n)$
  if $(r_j \neq 0)$
    $M' := \lfloor M/r_j \rfloor r_j$
  else
    $M' := M$
  $M'' := r_{j-1} \bmod M'$
  $hole := 0;\ buffer[0,\ldots,r_j) := \mathbf{A}[0,\ldots,r_j)$
  for $i := 1$ **to** $\lfloor r_{j-1}/M' \rfloor$
    assert: $hole = (i-1)M'$
    $extra\_buffer[0,\ldots,M'-r_j) := \mathbf{A}[hole + r_j,\ldots]$
    for $k := 1$ **to** $L_j - 1$ **{** $SegmentMove;\ hole := T(hole)$ **}**
    assert: $T(hole) = (i-1)M' + r_j$
    $\mathbf{A}[hole,\ldots] := extra\_buffer[0,\ldots,M'-r_j)$
    $hole := hole + M' - r_j$
    $SegmentMove;\ hole := T(hole)$
  assert: $hole = \lfloor r_{j-1}/M' \rfloor M' = r_{j-1} - M''$
  $extra\_buffer[0,\ldots,M''-r_{j+1}) := \mathbf{A}[hole + r_j,\ldots]$
  for $i := 1$ **to** $L_{j-1} - 1$ **{** $SegmentMove;\ hole := T(hole)$ **}**
  assert: $T(hole) = N - M''$
  $\mathbf{A}[hole + r_{j+1},\ldots] := buffer[0,\ldots,r_j - r_{j+1})$
  for $i := 1$ **to** $L_j - L_{j-1}$ **{** $SegmentMove;\ hole := T(hole)$ **}**
  assert: $T(hole) = r_{j-1} - M'' + r_j$
  $\mathbf{A}[hole,\ldots] := extra\_buffer[0,\ldots,M''-r_{j+1})$
  $hole := hole + M'' - r_{j+1};$
  if $(r_{j+1} \neq 0)$
    for $i := 1$ **to** L_{j-1} **{** $SegmentMove;\ hole := T(hole)$ **}**
    assert: $T(hole) = r_j - r_{j+1}$
    $\mathbf{A}[hole,\ldots] := buffer[r_j - r_{j+1},\ldots r_j)$

Figure 2.9: The multiple-load blockwise dolphin algorithm.

The only problem left is what to do if no $0 < r_j \le M$ exists. The problem when $r_j = 0$ is that it leaves M' and q_{j+1} undefined, but this can be managed simply by defining $M' = M$ if $r_j = 0$. We have the identity $\lfloor r_{j-1}/M' \rfloor = \lfloor q_{j+1} r_j / M' \rfloor$ when $r_j \neq 0$, which allows the removal of all occurrences of q_{j+1} from the program. The resulting program is presented in Figure 2.9.

Performance analysis

Each element is either moved directly to its final destination, causing two memory references per element, or moved via the buffer area, causing four memory references per element, so the total number of memory references executed is bounded by 4N. Indeed, if $\min\{m,n\} \le M'$, the number of elements moved via the buffer is $N(M' - \min\{m,n\})/M'$ which is close to N if $\min\{m,n\}$ is small, so it seems that the blockwise dolphin is no better than the reversal method or the swapping segments method.

Anyway, in the presence of a cache things looks somewhat differently. To produce an upper bound on the number of cache misses, we begin by citing a result by Sleator and Tarjan [ST85] showing that the LRU replacement strategy cannot be much worse than the optimal longest-forward-distance strategy (OPT), if the capacity of the LRU-cache is a bit larger than that used by the optimal strategy.

Lemma 2.1 [ST85] *Let $f_X(s)$ denote the number of block transfers performed by replacement strategy X for the sequence of memory references s, with a cache of size m_X measured in blocks. For any sequence s of memory references is*

$$f_{LRU}(s) \leq \frac{m_{LRU}}{m_{LRU} - m_{OPT} + 1} f_{OPT}(s)$$

if the initial contents of LRU's cache is the same as that of OPT's cache.

The idea is to construct a replacement strategy for which the number of blocks transfered from memory is close to the lower bound $2N/B$ and then use Lemma 2.1 to produce an upper bound on the number of cache misses incurred if the replacement strategy is LRU.

To begin with, we observe that we can make the buffer fit the small cache, by selecting $M' < M$, and then keep the buffer locked in the cache at all times. Further we set aside two cache lines for use as buffers during segment-moves, so that a segment-move of size k will incur at most the minimal number $2(\lceil k/B \rceil + 1)$ of cache misses.

The next step is to bound the number of segment-moves executed. The loop in Figure 2.9, line 9 will handle $\lfloor r_{j-1}/M' \rfloor L_j$ segments of size M'. The rest of the program will execute at most $L_j + L_{j-1}$ segment-moves. Since $\lfloor r_{j-1}/M' \rfloor L_j \geq L_j > L_{j-1}$ it is clear that on the average a segment-move must move at least $M'/3$ elements. Since each element is moved only once, we can conclude that in total at most $3N/M'$ segment-moves are executed. Hence the program will incur at most $2N/B + 6N/M' + O(1)$ cache misses when the buffer is fixed in the cache.

We can always select an M' such that $M/2 < cM' < M$ for any integer $c > 1$. We now apply Lemma 2.1 with $m_{OPT} = M'/B$ and $m_{LRU} = M/B$ and get that the number of cache misses incurred by an LRU-cache of size M is bound by

$$\begin{aligned} & \frac{M/B}{M/B - M'/B + 1}(2N/B + 6N/M' + O(1)) \\ \leq\; & \frac{M/B}{M/B - M/cB}(2N/B + 6N/M' + O(1)) \\ =\; & \frac{c}{c-1}(2N/B + 6N/M') + O(1) \end{aligned}$$

To summarize, we find that spatial locality suggests that a large buffer should be used, while temporal locality suggests the use of a small buffer size. Anyway, typical cache configurations are large enough to allow a reasonable choice to be made. As an example consider a cache configuration with $M = 16384$ and $B = 16$. If we, somewhat arbitrarily, select $c = 8$, so that the buffer size is $M/16 < M' \leq M/8$, we get that the number of cache misses is bound by $(8/7)(2N/B + 6N/(M/8)) + O(1) = 2.3N/B + O(1)$.

2.5 Implementation details and experimental results

To validate the theoretical findings, we carried out a collection of experiments with the programs available in the SGI STL and our tuned programs. The SGI STL provides an overloaded function template:

```
void rotate(Iterator first, Iterator middle, Iterator beyond)
```

which implements the three inplace methods described in Section 2.3. The methods are provided as separate implementations for random-access iterators, for bidirectional iterators and for forward iterators, automatically selected by the overload resolution mechanism of C++, based on the capability of the parameters. Our two tuned methods was implemented to be plug-in replacements for the SGI STL bidirectional iterator and random-access iterator implementation respectively, and further we also implemented the copying method described in Section 2.3.

In all of our experiments, input was provided to the SGI STL in the form of pointers to an array. Since pointers are random access iterators the STL would normally select the dolphin method for such an input, but we forced the STL to choose the other implementations by bypassing the standard call-interface. As input data we used 4-byte integers. In our experiments the sizes of the input were: 8 192, 262 144, and 8 388 608. The smallest problems fitted easily in the first-level cache, whereas the medium-size problems were too large to fit there, but they fitted in the second-level cache, and finally the largest problems were too large for the second-level cache, but small enough to fit in the main memory. For each of the three problem sizes, two sets of rotation values, m, were used. In the one set the numbers were designed specifically to make $\gcd(m, n)$ large by selecting m to be a power of two, while in the other set m was chosen to be the multiples of a prime close to $N/100$ thereby causing $\gcd(m, n)$ to be small. For every problem instance, the experiment was repeated $2^{23}/N$ times, each time with a new input array to ensure that no part of the array was already in the small cache when timing was started.

The experiments were repeated in several different computers with a hierarchical memory system. The results were similar in all the computers we tried, so we report here only those obtained on a Compaq AlphaServer DS20. The characteristics of this computer are as follows: It has two 500 MHz Alpha cpus (21264),- only one is used by our experiments,- 64 kbyte on-chip 2-way set associative cache with a line-size of 64 bytes, and a direct mapped 4 Mbyte off-chip cache, also with a cache-line size of 64 bytes. The experiments were carried out on dedicated hardware, so no cache interference from other sources was expected.

The programs were compiled by using the Digital C++ V6.1-027 compiler provided by the hardware manufacturer. The object codes were generated with the following command line options `-fast -arch ev6 -tune ev6 -unroll 8` controlling compiler optimizations, and the options `-newcxx -std ansi` specifying the variant of the programming language.

First, we tested the performance of the segment-interchanging programs described in Section 2.3, i. e. the three inplace programs available in the SGI STL and our own implementation of the copying method. Figures 2.10, 2.12, and 2.14 report the behaviour of these programs when the problem size is small, medium, and large, respectively. The reversal method is clearly the fastest of the inplace methods, with the non-inplace copying method running at roughly the same speed. Also the successive-segment-swapping method is only about two times slower. For both the reversal method and the successive-segment-swapping method we observe that there is very little degradation, about 10 percent, when the size of the cache is exceeded, suggesting that this particular memory subsystem is able to keep up with the cpu for sequential memory access. It can also be seen that the dolphin method is really slow when the problem does not fit the cache. On the largest inputs it is about 20 (twenty) times slower than the reversal method.

Second, we compared the performance of the best SGI STL program to the tuned versions of the reversal method and that of the dolphin method. The results obtained are reported in Figures 2.11, 2.13, and 2.15, again for the three different problem sizes. We see that when the problem fit the small cache, all three programs runs at about the same speed, with a slight advantage to the tuned three reversals method, while on the larger inputs our blockwise-dolphin method is about two times faster than the best SGI STL program. In general, the performance of the tuned reversal method is between the two, but due to cache interference the curves have some peaks when the size of segments to be interchanged are a large power of two, in which cases the method is the slowest of the three. Also it should be noted that the blockwise-dolphin algorithm uses less time per element on the larger inputs, suggesting that there is a non-trivial overhead for small inputs. To confirm this we ran the three reversals program, the tuned three reversals program, and the blockwise dolphin program with $N = 4, m = 2$. The result was that the tuned three reversals program was 1.6 times faster than the SGI STL three reversals program which in turn was 5.3 times faster than the blockwise dolphin program.

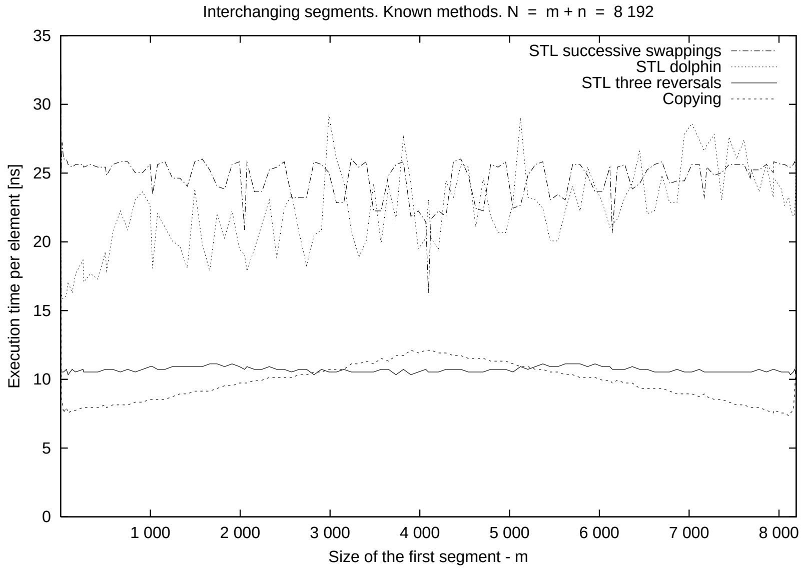

Figure 2.10: Execution time per element for the STL programs on a Compaq AlphaServer DS20 when $N = 8\,192$, m varies, and $n = N - m$.

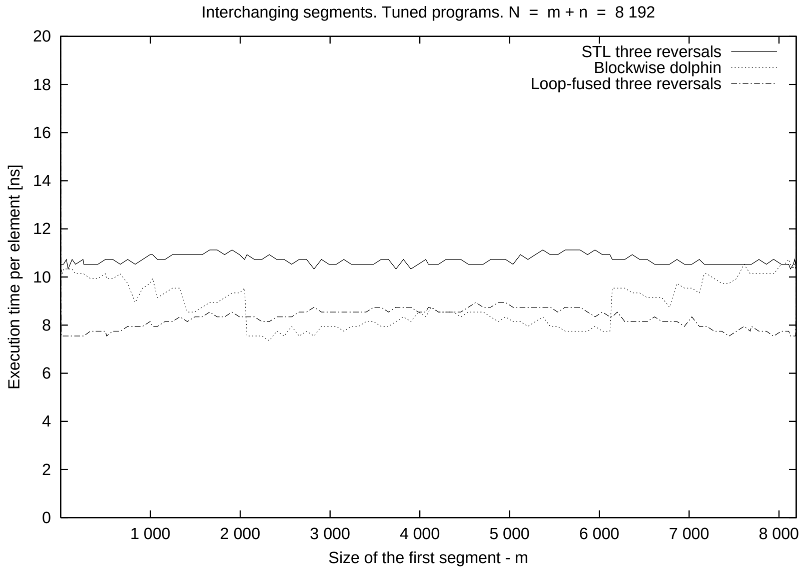

Figure 2.11: Execution time per element for the best STL program and our tuned programs on a Compaq AlphaServer DS20 when $N = 8\,192$, m varies, and $n = N - m$.

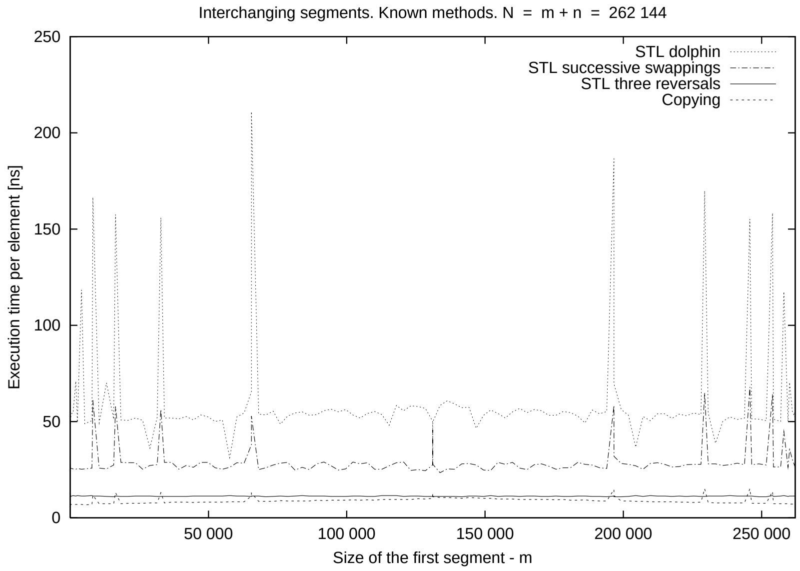

Figure 2.12: Execution time per element for the STL programs on a Compaq AlphaServer DS20 when $N = 262\,144$, m varies, and $n = N - m$.

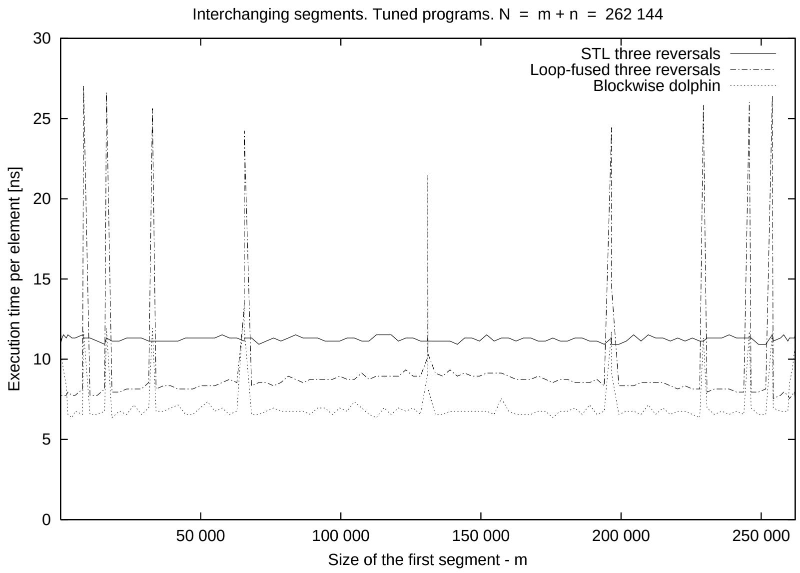

Figure 2.13: Execution time per element for the best STL program and our tuned programs on a Compaq AlphaServer DS20 when $N = 262\,144$, m varies, and $n = N - m$.

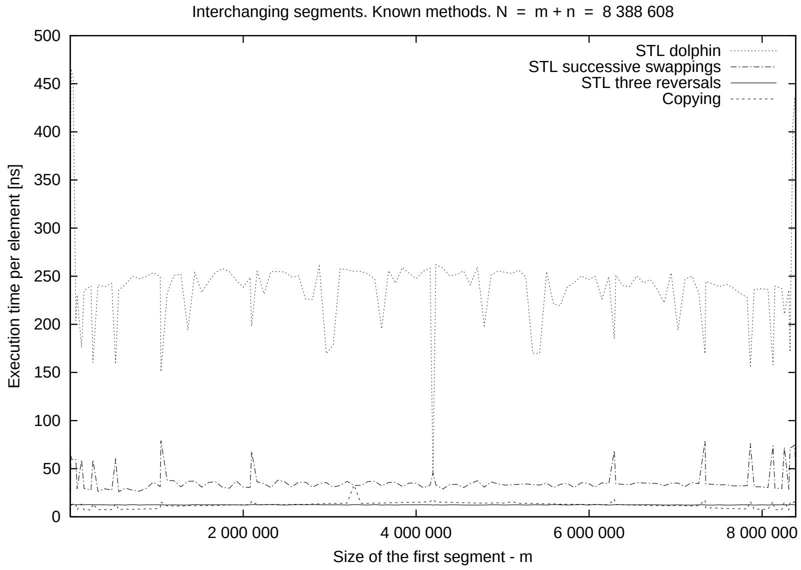

Figure 2.14: Execution time per element for the STL programs on a Compaq AlphaServer DS20 when $N = 8\,388\,608$, m varies, and $n = N - m$.

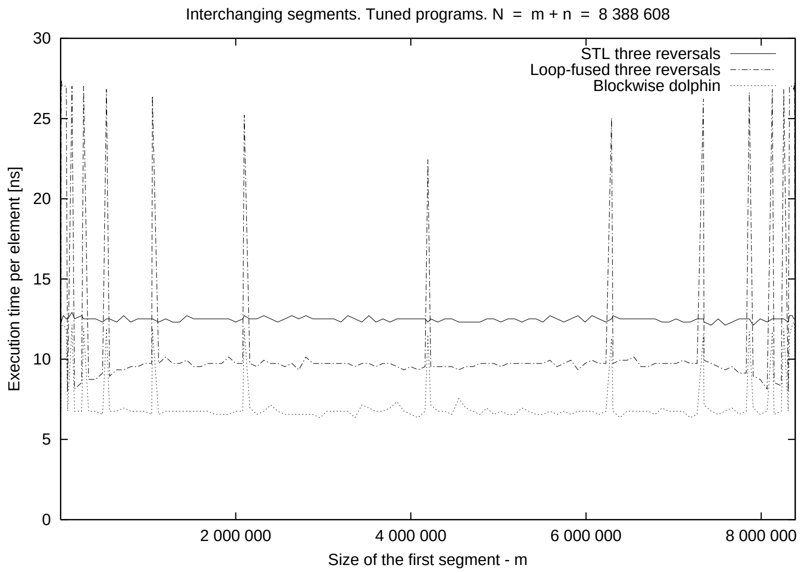

Figure 2.15: Execution time per element for the best STL program and our tuned programs on a Compaq AlphaServer DS20 when $N = 8\,388\,608$, m varies, and $n = N - m$.

2.6 Concluding remarks

Two new methods for rotating an array were developed. One is a simple modification of the well-known three-reversals algorithm using bidirectional iterators, and one is a complicated adaption of the dolphin method using random-access iterators. Our new methods were measured to be $1.5 - 2$ times faster than the best SGI STL implementation. However our main concern has been, not so much to tune array rotation but, to understand how to model a hierarchical memory in a way that is simple enough to be handled during the analysis and program development, while at the same time complete enough to capture any important performance characteristics of algorithms. It is encouraging to see that while our cache model makes a number of unrealistic assumptions to ease analysis, observed and predicted performance agrees very well. It should be mentioned though, that if the programs were compiled with less than full optimization, we would benefit less from our memory tuning.

The problem of rotating an array is quite simple, and normally one would not expect that much could be gained by program tuning. Nevertheless we see that our best implementation of an array rotation is more than 30 times faster than the equivalent implementation in the SGI STL on an important class of inputs. This is of course not the result of any substantially new ideas on our behalf, but is the result of an oversight in the SGI STL implementation. Indeed half of that speedup can be achieved simply by deleting the SGI STL specialization for random-access iterators and letting the SGI STL bidirectional-iterator implementation handle random-access iterators as well. It is a mistake to assume that a random-access iterator can be dereferenced in constant bounded time with a *small* constant. In the presence of a hierarchical memory system the constants involved can be substantial, and should not be ignored by a widely used and highly important library such as the STL. It should be emphasized that any cache model which measures spatial locality would recognise the problem with the dolphin algorithm, so it is rather surprising that it was included in the SGI STL in the first place.

Acknowledgements

The second author would like to thank the students who participated the performance-engineering courses held by him in Turku, Copenhagen, and Lund; and got the segment-interchanging problem as their home exercise. As we have seen, the exercise was more difficult than it seemed at first sight.

References

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[Ben86] Jon Bentley. *Programming Pearls.* Addison-Wesley, Reading, 1986.

[DD81] K. Dudziński and A. Dydek. On stable minimum storage merging algorithm. *Information Processing Letters*, 12:5–8, 1981.

[Du88] S. Dvořák and B. Ďurian. Merging by decomposition revisited. *The Computer Journal*, 31:553–556, 1988.

[FS66] W. Fletcher and R. Silver. Interchange of two blocks of data. *Communications of the ACM*, 9:326, 1966.

[GKP99] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 1999.

[GM81] David Gries and Harlan Mills. Swapping sections. Technical report, Department of Computer Science, Cornell University, Ithaca NY 14853, 1981.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, 2nd edition, 1996.

[Knu93] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley, Reading, 1993.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley Longman, Reading, 3rd edition, 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley Longman, Reading, 2nd edition, 1998.

[MS87] John L. Mohammed and Carlos S. Subi. An improved block-interchange algorithm. *Journal of Algorithms*, 8:113–121, 1987.

[She97] Ching-Kuang Shene. An analysis of two in-place array rotation algorithms. *The Computer Journal*, 40(9):541–546, 1997.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 3rd edition, 1997.

Chapter 3

Performance engineering case study: heap construction

joint work with Jyrki Katajainen and Maz Spork

Abstract. The behaviour of three methods for constructing a binary heap is studied. The methods considered are the original one proposed by Williams [1964], in which elements are repeatedly inserted into a single heap; the improvement by Floyd [1964], in which small heaps are repeatedly merged to bigger heaps; and a recent method proposed, e. g., by Fadel et al. [1999] in which a heap is built layerwise. Both the worst-case number of instructions and that of cache misses are analysed. It is well-known that Floyd's method has the best instruction count. Let N denote the size of the heap to be constructed, B the number of elements that fit into a cache line, and let c and d be some positive constants. Our analysis shows that, under reasonable assumptions, repeated insertion and layerwise construction both incur at most cN/B cache misses, whereas repeated merging, as programmed by Floyd, can incur more than $(dN \log_2 B)/B$ cache misses. However, for our memory-tuned versions of repeated insertion and repeated merging the number of cache misses incurred is close to the optimal bound N/B.

3.1 Introduction

In many research papers on sorting problems, the performance of competing methods is compared by analysing the number of element comparisons. However, in the case of small elements an element comparison can be as expensive as, for instance, an index comparison. Hence, even though the number of other operations involved is proportional to that of element comparisons, the latter measure alone does not give the whole truth about the actual performance. Moreover, to save some element comparisons complicated index manipulation might be necessary, and as a result the methods tuned only with the number of element comparisons in mind are often impractical (cf., [Car92]).

Another tradition popularized by Knuth (see, e. g., [Knu98]) is to analyse meticulously the number of all primitive operations performed. However, in current computers with a hierarchical memory the operations involving memory accesses can be far more expensive than the operations carried out internally in the processing unit. To predict accurately the execution time of a program, different costs must be assigned to different primitive operations. In particular, a special attention must be laid when assigning the cost for the memory access operations.

Most high-performance computers have several processors and a hierarchical memory, each memory level with its own size and speed characteristics. The different memory levels existent in the vast majority of computers are from the fastest and smallest to the slowest and largest: register file, on-chip cache, off-chip cache, main memory, and disk storage. The basic idea is to keep the regions of recently accessed memory as close to the processors as possible. If the memory reference cannot be satisfied by one memory level, i. e., in the case of a **miss** or a **fault**, the data must be fetched from the next larger memory level. Not only one datum is fetched, but a **block** of data in the hope that the memory locations in the vicinity of the current one will be accessed soon. In real programs a significant proportion of the running time can be spent waiting for the block transfers between the different levels of the memory hierarchy.

In this paper we study, both analytically and experimentally, the performance of programs that construct a binary heap [Wil64] in a hierarchical memory system. Especially, we consider the largest memory level that is too small to fit the whole heap. We call that particular level simply the **cache**. It should, however, be emphasized that our analysis is valid for the memory levels below this cache as well, provided that all our assumptions are fulfilled. We let B denote the **size** of the **blocks** transferred between the cache and the memory level above it, and M the **capacity** of the **cache**, both measured in elements being manipulated.

Recent research papers indicate that a binary heap is not the fastest priority-queue structure (see, e. g., [LL96, San99]) and heap-sort based on a binary heap is not the fastest sorting method (see, e. g., [MS91, LL99]), but we are mainly interested in the performance-engineering aspects of the problem. The reason for choosing the heap construction as the topic of this case study was the existence of the elegant Algol programs published by Williams [Wil64] and Floyd [Flo64]. In our preliminary experiments we translated these old programs into C++ and compared them to the C/C++ programs published in textbooks on algorithms and/or programming. It turned out that none of the newer variants were able to beat the old programs. Hence, the tuning of these old programs seemed to be a challenge.

It is well-known that the worst-case behaviour of Williams' method is bad, but for a randomly generated input Williams' program appears to be faster than Floyd's program when the size of the heap exceeds that of the cache. There are two reasons for this:

1. In the average case the number of instructions executed by Williams' program is linear [HM91], which is guaranteed in the worst case by Floyd's program.

2. Williams' program accesses the memory more locally than Floyd's program, as shown experimentally by [LL96].

Let N denote the size of the heap being constructed. We prove analytically that, if $M \geq rB\lfloor \log_2 N \rfloor$ for some real number $r > 1$, Williams' program incurs never more than $(2r/(r-1))N/B + O(\log_2 N)$ cache misses. On the other hand, there exists an input which makes Floyd's program to incur more than $(1/2)N(\log_2 B)/B - (1/2)N/B$ cache misses if $M \leq (1/4)N$. However, by memory tuning, and this is true for both of these programs, the number of cache misses incurred can be reduced close to the optimal bound N/B if the size of the cache is reasonable.

Also, efficient external-memory algorithms are developed with a multilevel memory model in mind. There the main concern is to keep the number of page transfers between main memory and disk storage as low as possible. Such algorithms can be good sources for cache-efficient programs as observed, e. g., by Sanders [San99]. A fast external-memory algorithm for heap construction have been presented by Fadel et al. [FJKT99]. Based on their idea a method for constructing a binary heap is obtained that has nice theoretical properties, e. g., it performs well even if the cache can only hold a constant number of blocks, but in practice it could not compete with the reengineered version of Floyd's program.

The structure of the rest of the paper is as follows. The models of computation and cost used in the theoretical analysis are defined in Section 3.2. The results of a micro benchmark that give some motivation for the cost model are reported in Section 3.2.4, but it can be skipped in the first reading. The notation used is established in Section 3.3. Also, the heap-construction problem is defined formally there. All the heap-construction methods discussed are described as running programs using C++ in a form compatible with the Standard Template Library (STL) which is part of the ISO standard for C++. The details about the STL are encapsulated in Section 3.3.2 which can be skipped in the first reading. The known heap-construction methods are described and analysed in Section 3.4 and our reengineered versions in Section 3.5. The experimental results on the performance of the heap-construction programs are reported in Section 3.6. In principle, the theoretical and the experimental parts of the paper could be read in any order. The paper is closed with some concluding remarks in Section 3.7.

3.2 Meticulous analysis

In meticulous analysis, the term coined by Katajainen and Träff [KT97], the goal is to analyse the running time of a program as exactly as possible, including the constant factors. This style of analysis is well-known from Knuth's books (see, e. g., [Knu98]). In this section we define the models of computation and cost, under which we carry out a meticulous analysis of heap-construction programs.

3.2.1 Primitive operations

We use the standard **word random-access machine** (RAM) (see, e. g., [Hag98]) as our model of computation with the exception that the memory has several levels in addition to the register file. All computations are carried out in registers while there are special instructions for loading data from memory to registers and for storing data from registers to memory. The number of registers available is unspecified but fixed, and each of the registers can be used both as an index register and a data register.

We assume that the programs executed by the machine are written in C++ [Str97] and compiled to pure C [KT97]. Pure-C statements are similar in strength to machine instructions on a typical present-day load-store reduced-instruction-set computer (RISC). Let `x` be a symbolic name of an arbitrary register, `y` and `z` symbolic names of arbitrary registers or constants, `p` a symbolic name of an index register (or a pointer), and λ some label. A **pure-C program** is a sequence of possibly labelled statements that are executed sequentially unless the order is altered by a branch statement. The **pure-C statements** are the following:

1. **Memory-to-register assignments**, that is, load statements: "`x = *p`".
2. **Register-to-memory assignments**, that is, store statements: "`*p = y`".
3. **Register-to-register assignments**: "`x = y`".
4. **Unary arithmetic assignments**: "`x =` $\ominus$`y`", where $\ominus \in \{+, -, \tilde{}, !\}$.
5. **Binary arithmetic assignments**: "`x = y` $\oplus$ `z`", where $\oplus \in \{+, -, *, /, \%, \&, |, \hat{}, <<, >>\}$.
6. **Conditional branches**: "`if (y` $\triangleleft$ `z) goto` λ", where $\triangleleft \in \{<, <=, ==, >, >=, !=\}$.
7. **Unconditional branches**: "`goto` λ".

Technically it is a simple matter to translate normal C++ control structures into pure C, but even for simple programs the amount of detail can be overwhelming. To get a better control over the details we have made extensive use of macros and adopted a few programming conventions. As an example consider the function given in Figure 3.1(a). A corresponding function in pure C is given in Figure 3.1(b). To minimize the extra cost caused by the loop control, the loop is ended conditionally as recommended by Sedgewick [Sed78]. As a consequence of this it may be necessary to create several copies of the loop control code, and of the code to exit the macro. Often we want to be able to use the same macro in different contexts so our macros are coded using continuation passing style [Rey93]. The instructions to be executed upon return from the macro are passed as a parameter on the macro call. Apart from allowing us to use the macro in several places this also eases the calculation of the pure-C cost. The instructions in a continuation need to be counted only once; namely, in the place where they appear in the source code. Finally, the `cat` and `xcat` macros are borrowed from [KR88, p. 231]. We use them in combination with the preprocessor macro `__LINE__` to generate unique labels. This is important to prevent name clashes between macros and to allow a single macro to be used multiple times in the same function, even though the example program is too simple to need this.

3.2.2 Hierarchical memory model

Let w denote the **size** of the **memory words**, measured in bits. We assume that the **capacity** of **memory** is at most 2^w words, so that the address of any

```
void c(unsigned int n) {
  while (n > 1)
    if ((n & 1) == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
}
```

(a)

```
#define xcat(x, y) x ## y
#define cat(x, y) xcat(x, y)

#define LOOP_BODY(n, continuation)           \
  unsigned int parity = n & 1;               \
  if (parity != 0) goto cat(label, __LINE__); \
  n = n / 2;                                 \
  continuation;                              \
cat(label, __LINE__):                        \
  n = 3 * n;                                 \
  n = n + 1;                                 \
  continuation

void pure_c(unsigned int n) {
  if (n <= 1) return;
  loop:
    LOOP_BODY(n, if (n > 1) goto loop; return);
}
```

(b)

Figure 3.1: (a) A sample program and (b) its translation into pure C.

word can be stored in one word. The memory is arranged into several, say, ℓ levels abiding the principle of inclusion: if some datum is present at level i, it is also present at level $i+1$. The lowest memory level is the register file. Level $i+1$ is assumed to be larger and slower than level i. The contents of each memory level is divided into disjoint, consecutive **blocks** of fixed uniform size, but the block size at different levels may vary. When the block accessed, i. e., read or written, is not present at some level, i. e., when a **miss** occurs, it is fetched from the next larger level and one of the existing blocks is removed from the cache.

In general, we are mainly interested in the highest memory level that is too small to fit the input data. We call that particular level simply the **cache**. We let B denote the **size** of **blocks** used in the cache, and M the **capacity** of the **cache**, both measured in elements being manipulated. For the sake of simplicity, we assume that both B and M are powers of 2, and that $M \geq 2B$. Even if many cache-based systems employ set associative caches (for the technical details the reader is referred to the book by Hennessy and Patterson [HP96]), we assume that the cache is **fully associative**, i. e., when a block is transferred to the cache, it can be placed in any of the M/B cache lines. We assume that the block replacement is controlled by the cache system according to the **least-recently-used** (LRU) policy. In particular, the program has no control over which block is removed when a new block arrives.

3.2.3 Penalty cost model

We use a weighted version of the cost model presented in [KT97]. Alternatively, one can see the cost model as a simplification of the Fortran-based model discussed in [SS95].

1. The cost of all pure-C operations is assumed to be the same τ.
2. Each load and store operation has an extra penalty $\tau_i^\star$ if it incurs a miss at level i of the memory hierarchy, $i \in \{1, 2, \dots, \ell\}$.

Now, if a program executes n pure-C operations and incurs m_i misses at memory level i, $i \in \{1, 2, \dots, \ell\}$, its execution time might be expressed as the sum

$$n \cdot \tau + \sum_{i=1}^{\ell} m_i \cdot \tau_i^\star .$$

So, in addition to the normal instruction count, it is important to calculate the number of misses that occur at different memory levels.

This cost model allows computer-independent meticulous analysis of programs, but it is overly simplified. Some computer architectures may offer combinations of the pure-C operations as a single primitive operation, e. g., it might not be necessary to translate `x = a[i]` into the form `p = a + i`, `x = *p`; while others are unable to execute some pure-C operations as a single operation, e. g., it might only be possible to carry out a comparison to zero so `if (y > z) goto` λ has to be translated into the form `x = y - z`, `if (x > 0) goto` λ. In some computers, `x = y * z` may be more expensive than `x = y + z`, et cetera. In addition, the cost model disregards a series of latency sources such as pipeline hazards, branch delays, register spilling, contention on memory ports, and translation look-aside buffer (TLB) misses.

3.2.4 Practical considerations

Let τ denote the cost of each pure-C operation, and $\tau^\star$ the miss penalty for the cache closest to the main memory. The parameters τ and $\tau^\star$ are highly computer-dependent. In order to get an idea of the relationship between these two quantities we examined experimently the performance of the summing program given in Figure 3.2 for different *step* values on several computers.

This C program computes the sum (modulo the word size) of the N integers given in an array whose start address is **a**. The rationale behind the program is that for each value of N it executes the same sequence of instructions independently of the value of *step*, provided that *step* is a prime relatively to N. When *step* $= 1$ the program scans the integers in the array sequentially. When *step* is an odd number larger than the block size B, and the array size N is a power of

```
unsigned int sum(unsigned int* a, unsigned int step, unsigned int N) {
  // step must be an odd number, 0 < step < N.
  // N must be a power of 2.
  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;
}
```

Figure 3.2: Summing integers in a special way.

2, each location of the array is visited exactly once, but two subsequent memory references access different blocks in memory.

A simple analysis shows that each iteration of the `for`-loop executes 6 pure-C operations. We ran the summing program for the values $step = 1$ and $step = p$, p being the smallest prime larger than the block size B, and measured the execution times t_1 and t_p for $N = 2^{23}$. With these choices of $step$ the sequential scan incurs N/B cache misses, whereas the jumping scan leads to N cache misses at least if NB/p is larger than the capacity of the cache. Assuming that the execution times are completely determined by the parameters τ and $\tau^\star$, we get the equations

$$\begin{cases} 6N \cdot \tau + N/B \cdot \tau^\star & = & t_1 \\ 6N \cdot \tau + N \cdot \tau^\star & = & t_p \, , \end{cases}$$

from which we can solve τ and $\tau^\star$. Table 3.1 gives the obtained values for four different computers. These experiments were carried out by using the vendors own C compiler with full optimization.

Strictly speaking this micro benchmark does not accurately measure the ratio of the memory-access latency to the instruction-issue rate. Due to its predictable access pattern and because there are no data dependencies in the loop, it allows smart compilers/hardware to overlap index calculations and loop control with memory accesses. Also, since there are no data dependencies between loop iterations, a multibanked memory should be able to take full advantage of the parallelism in the memory system to hide some of the latency. However, the benchmark shows clearly that memory latencies can be considerable, and that it is worth to make memory access patterns as local as possible if a speed-up by a constant factor is of interest.

A micro benchmark similar to ours could be designed to measure other relevant performance characteristics of the memory hierarchy. An interested reader should consult, e.g., the study by Saavedra and Smith [SS95] which describes a collection of experiments that can be used to compute many memory hierarchy parameters, including the size of the cache(s) and the TLB, the time needed to satisfy a cache or TLB miss, and the cache and TLB associativity.

Table 3.1: Estimation of τ and $\tau^\star$ in four different computers.

Computer	Clock cycle	B	t_p/t_1	τ	$\tau^\star$
IBM RS/6000 34H[a]	23.8 ns	16	8.8	8.4 ns	880.4 ns
IBM RS/6000 397[b]	6.25 ns	32	6.3	2.9 ns	114.7 ns
IBM F50[c]	3 ns	16	10.1	1.8 ns	268.3 ns
Compaq AlphaServer DS20[d]	2 ns	16	3.3	2.8 ns	47.9 ns

[a] On-chip cache: none; Off-chip cache: 32 kbytes, 64 bytes per cache line, 4-way set associative.

[b] On-chip cache: 128 kbytes, 128 bytes per cache line, 4-way set associative; Off-chip cache: none.

[c] On-chip cache: 32 kbytes, 32 bytes per cache line, 4-way set associative; Off-chip cache: 256 kbytes, 64 bytes per cache line, 8-way set associative.

[d] On-chip cache: 64 kbytes, 64 bytes per cache line, 2-way set associative; Off-chip cache: 4 Mbytes, 64 bytes per cache line, direct mapped

3.3 Preliminary definitions

In this section we establish our notation and define the problem of constructing a heap formally.

3.3.1 Heap construction

Let $\mathcal{S}$ be a set of **elements** of arbitrary type. A binary relation $\bigcirc$ on $\mathcal{S}$ is **irreflexive** if $x \bigcirc x$ is false for all x in $\mathcal{S}$, and it is **transitive** if $x \bigcirc y$ and $y \bigcirc z$ implies $x \bigcirc z$ for all x, y, and z in $\mathcal{S}$. A binary relation $\circledless$ is called a **strict weak ordering** if it is irreflexive, transitive, and if the relation $\ominus$ defined by

$$x \ominus y \text{ if and only if both } x \circledless y \text{ and } y \circledless x \text{ are false}$$

is transitive. Informally, this means that two elements whose relative order is not determined by the relation $\circledless$ are considered to be equal (but not identical). Since a strict weak ordering behaves like the $<$ relation on the natural numbers, we say that x is **smaller than** y if $x \circledless y$. The **converse relation** $\circledgtr$ of relation $\circledless$ is defined by

$$x \circledgtr y \text{ if and only if } y \circledless x\,.$$

In a similar manner, the other inequality relations could be defined by means of this single strict weak ordering.

In the following the basic concepts related to binary trees are assumed to be known; for the definitions of the concepts used, but not defined, we refer to [Knu97, Section 2.3]. The **depth** or **level** of a **node** in a binary tree is the length of the path from that node to the root. That is, the root is on level 0, its children on level 1, and so on. The **height** of a **node** in a binary tree is the length of the longest path from that node to a leaf. The **height** of a **binary tree** is the maximum height (or depth) of any of its nodes. We call a binary tree **complete** if all its branch nodes (if any) have two children and all its leaves (if any) are on the same level. That is, a complete binary tree of height $h \geq 0$ must have $2^{h+1} - 1$ nodes. Furthermore, we call a binary tree **left-complete** if the tree can be obtained from a complete binary tree by removing some of its rightmost leaves. That is, the height of a left-complete binary tree with N nodes, $N \geq 1$, is $\lfloor \log_2 N \rfloor$.

For two integers i and k, $i \leq k$, we let $[i\,..k)$ denote the set $\{i, i+1, \ldots, k-1\}$, and $\mathbf{a}[i\,..k)$ an **indexed sequence** of $k-i$ elements that supports indexing. The **index** of each **element** is equal to i plus the number of elements that precede it in $\mathbf{a}[i\,..k)$, and we use $\mathbf{a}[j]$ to denote the element of $\mathbf{a}[i\,..k)$ with index j if j is in $[i\,..k)$. We adopt the convention that $\mathbf{a}$ denotes the **base position** of **indexed sequence** $\mathbf{a}[i\,..k)$. Using this convention, the **position** of **element** $\mathbf{a}[j]$ [which is equivalent to $*(\mathbf{a} + j)$ in C++] is $\mathbf{a} + j$. We say that j is a **valid index** for indexed sequence $\mathbf{a}[i\,..k)$ if j is in $[i..k)$, and that $\mathbf{p}$ is a **valid position** for indexed sequence $\mathbf{a}[i\,..k)$ if $\mathbf{p}$ is in $[\mathbf{a} + i\,..\mathbf{a} + k)$. A position like an element has an index: the **index** of **position p** with respect to base $\mathbf{a}$ is $\mathbf{p} - \mathbf{a}$.

A left-complete tree with N nodes, each storing an element, can be represented in an indexed sequence by storing the elements levelwise: first the element at the root, then the elements at the children of the root, and so on. We call such a representation of a left-complete tree an **implicit tree**. An implicit tree corresponding to an indexed sequence $\mathbf{a}[0..N)$ can be represented compactly and completely by the pair $(\mathbf{a}, \mathbf{a}+N)$. Another possibility is to use the pair $(\mathbf{a}-1, N)$ after which the root has index 1 with respect to base $\mathbf{a}-1$. Moreover, any subtree of that tree can be represented by the triple $(\mathbf{a}-1, j, N)$ or $(\mathbf{a}, \mathbf{p}, \mathbf{a}+N)$, where j ($\mathbf{p}$) is the index (position) of the root of the subtree.

In an implicit tree the parent-child relationships can be realized without storing any positional information at the nodes. In an **index-tied representation** of a node, keeping implicit that the root has index 1, the parent, left child, and right child of a node with valid index j have indices $\lfloor j/2 \rfloor$, $2j$, and $2j+1$ (if these are valid), respectively. In a **position-tied representation** of a node, if the root has index 1 with respect to base $\mathbf{a}$, the parent, left child, and right child of node at position $\mathbf{p}$ have positions $\mathbf{a}+\lfloor(\mathbf{p}-\mathbf{a})/2\rfloor$, $\mathbf{a}+2(\mathbf{p}-\mathbf{a}) = \mathbf{p}+(\mathbf{p}-\mathbf{a})$, and $\mathbf{a}+2(\mathbf{p}-\mathbf{a})+1 = \mathbf{p}+(\mathbf{p}-\mathbf{a})+1$ (if these are valid), respectively. An interface to a class, which represents a node by its index, is given in Figure 3.3. In a sense, this class extends the capabilities of indices with the normal parent-child relationships. A class, which represents a node as an extension of a position, is similar, but for efficiency reasons we have implemented it as a collection of functions. We use both of these node representations in our own programs.

```
// This class implements an index-tied node assuming implicitly that the root
// has index 1. Note that some of the functions require that the underlying
// hardware represents the indices in the two's complement form.

namespace index_tied_node {

  template<class index>
  class node {
  public:
    node();
    node(index);
    operator index () const;

    // Predicates

    bool is_valid(const node<index>) const;
    bool is_root() const;
    bool is_left_child() const;
    bool is_right_child() const;

    // Functions

    node<index> root() const;
    node<index> parent() const;
    node<index> left_child() const;
    node<index> right_child() const;
    node<index> left_sibling() const;
    node<index> right_sibling() const;
    node<index> successor() const;
    node<index> predecessor() const;
    node<index>& operator++(); // Only prefix ++ supported
    node<index>& operator--(); // Only prefix -- supported

  private:
    index current;
  };
};

#include "index_tied_node.cc"
```

Figure 3.3: A node class extending the capabilities of an index with the parent-child relationships.

A tree in which each node stores an element is a **heap** with respect to a strict weak ordering $\circledless$ if, for each branch of the tree, the element x stored in it is no smaller than the element y stored in any of its children, i. e., $x \circledless y$ must be false. Informally, we say that such a tree is a maximum heap, or simply a heap. In the **heap-construction problem** studied in this paper, we are given an implicit tree and a strict weak ordering, and the task is to transform this tree into a heap with respect to the strict weak ordering. If possible, we would like to carry out the heap construction **in-place**, i. e., by modifying the input tree and using only $O(1)$ additional memory words. We call the outcome of this transformation an **implicit heap**.

For the purpose of our analysis, we assume that the elements in the given implicit tree are stored consecutively in memory, and that the implicit tree is **perfectly aligned**, i. e., that the first position of the block containing the root of the tree is empty. That is, the first block consists of this empty position plus the first $B-1$ nodes of the input tree, each level of the tree with $2^i \geq B$ nodes is divided into exactly $2^i/B$ blocks, and at the last level the last block may again be nonfull. (Recall that we assumed B to be a power of 2.) Furthermore, we assume that a block can fit at least two elements, i. e., $B \geq 2$, and that a block only keeps whole elements. These assumptions guarantee that the children of a node will always be situated in the same block.

3.3.2 STL compatibility

We have implemented our programs in C++ in the form compatible with the routine `std::make_heap` available in the Standard Template Library (STL) which is part of the ISO standard for C++. This routine has two overloaded interfaces:

```
template <class position>
void make_heap(position first, position beyond);

template <class position, class ordering>
void make_heap(position first, position beyond, ordering less);
```

The parameters `first` and `beyond` are *iterator objects* that must support the operations: `operator*` (prefix), `operator->`, `operator[]`, `operator++` (prefix and postfix), `operator--` (prefix and postfix), `operator+` (add an integer to a position), `operator-` (subtract an integer from a position and calculate the difference of two positions), `operator+=`, `operator-=`, `operator==`, `operator!=`, `operator<`, `operator>`, `operator>=`, and `operator<=`. According to the STL terminology these objects are random-access iterators. The parameters `first` and `beyond` specify the position of the first element stored at the root of the given implicit tree and that of the first artificial element beyond the implicit tree, respectively. The first version of `std::make_heap` uses `operator<` provided for the elements in the comparisons, whereas the second one accepts any *function object* `less`, which supports the operation `operator()`, as its third parameter. In effect, `operator<` for the elements or `operator()` for the function object `less` must define a strict weak ordering on the set of elements.

One complication caused by this interface is that we only know the type of the positions, but not the type of the elements stored at these positions. To dereference a position and manipulate the resulting element, or to operate with the difference of two positions, we utilize the `std::iterator_traits` facility [Str97, Section 19.2.2] provided by the STL.

The routine `std::make_heap` of the STL (version 3.2) implements Floyd's method (see Section 3.4.2) with the sift-hole-down-sift-up heuristic [Knu98, Exercise 5.2.3.18], which saves some comparisons [MR89], but in practice the heap construction becomes slower due to an increase in the number of element moves.

3.4 Known methods and their analysis

In this section, we recall and analyse meticulously the heap-construction methods presented by Williams [Wil64], Floyd [Flo64], and Fadel et al. [FJKT99]. Both the number of cache misses incurred and the number of instructions performed are analysed. The instruction counts are measured in pure-C operations under the assumptions that the elements are one-word integers and that the given implicit tree is stored in an array.

3.4.1 Repeated insertion

In the repeated-insertion heap-construction method proposed by Williams [Wil64], the nodes of the input tree are visited in the order of their indices and the element at each new node is inserted into the implicit heap constructed so far. In the basic operation, often called **sift-up**, the tree is traversed upwards starting from the new node, and moving the element stored at the parent of the current node down until the proper position for the new element is found, i. e., when the root is reached or when an element stored at the parent is not smaller than the new element. In Figure 3.4 a C++ implementation of the sift-up function and in Figure 3.5 that of the heap-construction method are given very much in the spirit of Williams' original Algol program. The loops are organized with `goto` statements, since Algol 60 did not support any other loop construct.

```
// The sift-up function as programmed by Williams.

#include <iterator> // defines std::iterator_traits

namespace Williams {

  template<class position, class index, class ordering>
  inline void sift_up(position a, index k, ordering less) {
    typedef std::iterator_traits<position>::value_type element;
    index j = k;
    element in = a[j];
  scan:
    if (j > 1) {
      index i = j / 2;
      if (less(a[i], in)) {
        a[j] = a[i];
        j = i;
        goto scan;
      }
    }
    a[j] = in;
  }
}
```

Figure 3.4: The sift-up function in C++.

```
// Construct a heap using the repeated-insertion method.
// This program is a translation of Williams' original Algol program into C++.

#include <iterator>   // defines std::iterator_traits
#include "sift_up.cc" // defines the sift-up function

namespace Williams {

  template<class position, class ordering>
  void make_heap(position first, position beyond, ordering less) {
    typedef std::iterator_traits<position>::difference_type index;
    const index N = beyond - first;
    if (N < 2) return;
    const position a = first - 1;

    index j = 1;
  L:
    j = j + 1;
    sift_up(a, j, less);
    if (j < N) goto L;
  }
}
```

Figure 3.5: Williams' heap-construction program in C++.

In the worst case, each level of the heap is visited exactly once in the inner loop (the `scan` loop in `Williams::sift_up`). One element comparison is done at each level, except the bottommost one, so the number of element comparisons performed is equal to the height of the heap. Since the height of an implicit heap of size j is $\lfloor \log_2 j \rfloor$, the construction of an N-element heap requires $\sum_{j=2}^{N} \lfloor \log_2 j \rfloor$ element comparisons, which is at most $N \log_2 N - 1.91N + O(\log_2 N)$ element comparisons. (For the derivation of this sum, see, e.g., [Weg92].) By simple transformations the inner loop is seen to execute 9 pure-C operations, 4 of which are executed prior to the element comparison. After inlining `Williams::sift_up`, the outer loop (the `L` loop in `Williams::make_heap`) executes 7 pure-C operations. Hence, the pure-C operation count is at most $7N + 9\,(N \log_2 N - 1.91N + O(\log_2 N))$, which is bounded by $9N \log_2 N - 10N + O(\log_2 N)$.

Hayward and McDiarmid [HM91] proved that, under the assumptions that all the N elements stored in the initial tree are distinct and that each placement of the elements into the nodes is equally likely, the number of promotions performed, i.e., assignments `a[j] = a[i]` in the inner loop, is on an average at most 1.3. By using this we get that in the average case the number of pure-C operations performed is less than $7N + 9 \cdot 1.3N + 5N + O(1)$, which is bounded by $24N + O(1)$.

Next, we analyse the cache performance of Williams' heap-construction program. The cache-miss rate is in direct relation to the capacity of the cache. If the cache is small, the number of cache misses incurred may be the same as that of memory reads, which is equal to $N - 1$ plus the number of element comparisons. The memory writes occur always in the close temporal proximity of reads, so they will not incur any cache misses. However, we show that for a cache of reasonable size the cache performance is only at most a constant factor from the optimal bound N/B. On the other hand, our lower-bound results show that the number of cache misses cannot get arbitrary close to the optimal bound.

We begin by introducing some terminology. The node indicated by parameter `k` of `Williams::sift_up` is called the **new node**. Furthermore, a block is said to be **relevant** to node v if it contains a node that is on the path from v to the root of the initial tree. So, if a block relevant to the new node is in the cache, it is undesirable to remove it from there since it might be accessed later on.

Let s be any sequence of memory references. If A is a replacement strategy, we denote by m_A the capacity of the cache used by A, measured in blocks, and by $f_A(s)$ the number of cache misses incurred by A on s. The main ingredient in many of our upper-bound proofs is the following result by Sleator and Tarjan [ST85] showing that the LRU replacement strategy cannot be much worse than the optimal longest-forward-distance (OPT) replacement strategy if the capacity of its cache is a bit larger than that used by the optimal strategy.

Lemma 3.1 [ST85] *For any sequence s of memory references,*

$$f_{LRU}(s) \leq \frac{m_{LRU}}{m_{LRU} - m_{OPT} + 1} f_{OPT}(s)$$

if the initial contents of LRU's cache is the same as that of OPT's cache.

Now we are ready to prove both an upper bound and a lower bound on the worst-case cache performance of Williams' heap-construction program. We also prove a lower bound on its best-case cache performance.

Theorem 3.1 *Let B denote the size of cache blocks, M the capacity of the cache, and N the size of the heap to be constructed. Assume that initially the cache is empty and that the block-replacement strategy is LRU.*

An upper bound for the worst case. *Assuming that the cache can hold at least $r\lfloor \log_2 N \rfloor$ blocks for some real number $r > 1$, Williams' heap-construction program incurs at most $(2r/(r-1))N/B + O(\log_2 N)$ cache misses.*

A lower bound for the worst case. *There exist an input of size $N \geq 2M$ for which Williams' heap-construction program incurs at least $2N/B - 2M/B$ cache misses.*

A lower bound for the best case. *For any input Williams' heap-construction program will incur at least $\max\{N/B, (3/2)N/B - M/B - 1\}$ cache misses.*

Proof: The upper bound for the worst case. Let s be the sequence of memory references produced by Williams' heap-construction program for an arbitrary input of size N. The idea is to construct a block-replacement strategy A which guarantees that $f_A(s) \leq 2N/B + O(\log_2 N)$ if $m_A = \lfloor \log_2 N \rfloor + 1$. Since $f_{OPT}(s) \leq f_A(s)$, if $m_{OPT} = m_A$, we can use Lemma 3.1 to conclude that, for a real number $r > 1$, $f_{LRU}(s) \leq (2r/(r-1))N/B$ if $m_{LRU} = M/B \geq r\lfloor \log_2 N \rfloor$.

To complete the proof we present the principle of the replacement strategy A. The basic observation is that, when visiting the nodes at one level, each of the blocks holding a node from this particular level and any of the previous levels will become relevant to the new node, remain relevant for a while, after which it is no more relevant (until processing the next level). Even if a block is relevant to the new node, it need not be accessed at all. Hence, we let A keep the blocks relevant to the new node in the cache at all times. In particular, the functioning of A does not depend on the blocks accessed, but only on the index of the new node. When some block becomes relevant to the new node, its predecessor holding nodes from the same level is removed from the cache after which the block is transferred to the cache.

Let us now calculate the number of cache misses incurred when the replacement strategy A is in use under the assumption that the cache can fit $\lfloor \log_2 N \rfloor + 1$ blocks. Further, assume that $2^{t-1} \leq N < 2^t$ for some positive t, and let $K = N - 2^{t-1} + 1$. Of course, the insertions of the $B - 1$ elements on the $\log_2 B - 1$ topmost levels incur only one cache miss. The insertion of the 2^ℓ elements at level ℓ, $\log_2 B \leq \ell < t - 1$, incurs at most $2^{\ell+1}/B - 1$ cache misses, and the insertion of the K elements at level $t - 1$ can incur at most $\sum_{i=0}^{\lfloor \log_2 N \rfloor} \lceil K/(2^i B) \rceil$ cache misses. By summing over all levels, we get the upper bound $2N/B + O(\log_2 N)$ for the number of cache misses incurred.

The lower bound for the worst case. Assume that $N = 2^t - 1$ for some integer t and that $N \geq 2M$. As a bad input, we construct a complete implicit tree of size N as follows. For each level ℓ, $0 \leq \ell \leq \log_2 B$, the elements stored at the nodes there are all equal to some fixed element, called a **stopper**. For each level ℓ, $\log_2 B < \ell < \log_2 M$, the elements stored are divided into two groups of equal size. In the first half the elements are in ascending order and all are smaller than the stopper, but they are all larger than the elements among the first group of elements at level $\ell - 1$, except for $\ell = \log_2 B + 1$. In the second half the elements are also in ascending order, and they are all larger than any element stored at the previous levels. Finally, for each level ℓ, $\log_2 M \leq \ell < t$, the elements stored are again divided into two groups. The size of the first group is $M/2$ and that of the second group $2^\ell - M/2$. As earlier, the elements in the first group are in ascending order, except some special elements, all smaller than the stopper, but still larger than the elements among the first group at the previous level. There are B **special elements**. They are all equal to the stopper and they are stored regularly so that every $M/(2B)$th element in the first group is a special element. The second group of elements is similar to the corresponding group at the previous levels: they are given in ascending order and all are larger than the elements at the previous levels.

The crux of this construction is that, for each level ℓ, $\log_2 M < \ell < t$, when processing the first group of elements stored at that level none of the blocks needed are among the blocks fetched in when processing the last $M/2$ elements at level $\ell - 1$. Moreover, when processing the second group of elements at level ℓ, $\log_2 M < \ell < t$, none of the blocks needed are among the blocks fetched in when processing the first group of elements at that level, except one block containing stoppers. Moreover, the special elements fill up one block at the top with stoppers, so there will also be stoppers for the sift-up operations done for the first group of elements at the next level. For a further illustration of the construction, see Figure 3.6.

During processing the first group of elements all the blocks $\log_2 M - 1$ levels upwards has to be visited, but the stoppers prevent the sift-up operations to proceed any further. In total, M/B blocks are accessed. So when the processing of the second group of elements starts, none of the blocks used when processing

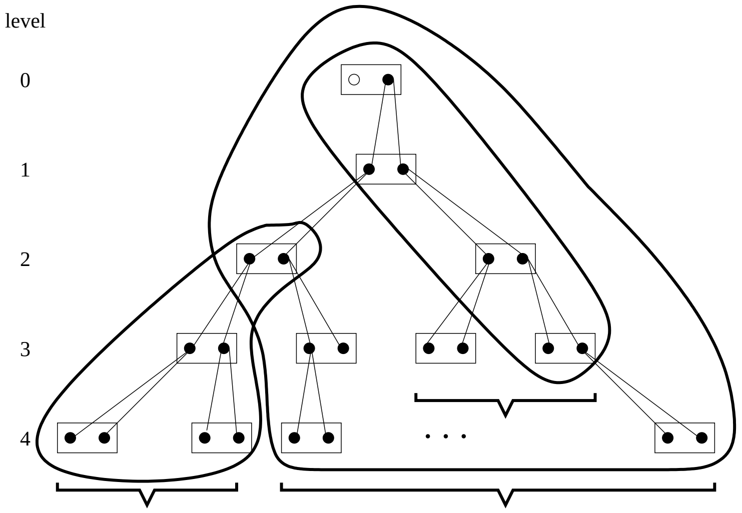

Figure 3.6: Illustration of the proof of the lower bound for $B = 2$, $M = 8$, and $N = 31$. The blocks inside the cache after processing the last $M/2$ elements at level 3, those after processing the first group of elements at level 4, and those fetched in when processing the second group of elements at level 4 are circled.

the elements from the previous level are in the cache any more. For all the elements in the second group the sift-up operations proceed always up to the root. Hence, when handling the elements at level ℓ, $\log_2 M < \ell < t$, all the blocks occupying these elements and the elements at all the previous levels must be fetched in at least once.

Let us now analyse the number of cache misses incurred. Clearly, to handle the first $2M - 1$ elements at least $2M/B$ cache misses are incurred. There are 2^ℓ elements stored at level ℓ and $2^{\ell+1} - 1$ elements altogether up to the level ℓ. The nodes containing these $2^{\ell+1} - 1$ elements occupy $2^{\ell+1}/B$ blocks and all these have to be fetched into the cache at least once when processing the nodes at level ℓ. To sum up, the number of cache misses incurred is at least

$$2\frac{M}{B} + \sum_{\ell=\log_2 M+1}^{t-1} \frac{2^{\ell+1}}{B},$$

which is bounded from below by $2N/B - 2M/B$.

The lower bound for the best case. Consider any node of the given implicit tree of size N that is the first node in its corresponding block. Clearly, a cache miss is incurred when the element stored at this node is inserted into the existing heap. Assume now that $N \geq 2M$. Further, assume that v is a branch, whose index is larger than or equal to M, and which is the first node in its block. If the index of v is j, the index of its left child is $2j$. Hence, after inserting the element at v into the heap, and before inserting the element at its left child, at least M elements have been added to the heap occupying at least M/B blocks. In the insertions here between, the first B insertions access the block that contained v, but hereafter none of the other sift-up operations access

```
// The sift-down function as programmed by Floyd.

#include <iterator> // defines std::iterator_traits

namespace Floyd {

  template<class position, class index, class ordering>
  inline void sift_down (position a, index i, index N, ordering less) {
    typedef std::iterator_traits<position>::value_type element;
    index j;
    element copy = a[i];
  loop:
    j = 2 * i;
    if (j <= N) {
      if (j < N)
        if (less(a[j], a[j + 1]))
          j = j + 1;
      if (less(copy, a[j])) {
        a[i] = a[j];
        i = j;
        goto loop;
      }
    }
    a[i] = copy;
  }
}
```

Figure 3.7: The sift-down function in C++.

that block any more. When the element at the left child of v is to be processed, the block containing v is no more in the cache incurring a cache miss since each sift-up operation should at least check the relation between the element stored at the new node and the element stored at the parent of the new node.

To summarize, the insertion of an element at each of the at least N/B nodes, which is the first one in its corresponding block, incurs a cache miss. Moreover, the insertion of an element at the left child of each of the at least $(N - \lceil N/2 \rceil - M)/B$ branches, which is the first one in its corresponding block and whose index is larger than M, incurs an extra cache miss. Thus, in this particular case the number of cache misses incurred is at least $(3/2)N/B - M/B - 1$.

3.4.2 Repeated merging

In the repeated-merging heap-construction method proposed by Floyd [Flo64] small heaps are merged to form larger heaps. The basic operation, called **sift-down**, transforms a subtree rooted by a node into a heap if the subtrees rooted by the children are both heaps. In Floyd's scheme the nodes are considered in the increasing order of heights. Trivially, all singleton nodes form a heap. Therefore, the processing starts from the last branch, and proceeds backwards until the root is reached and the sift-down operation for it executed. Figure 3.7 gives a C++ implementation of the sift-down function and Figure 3.8 that of this heap-construction method in a form similar to Floyd's original Algol program.

Let h_i denote the height of the node with index i in the given implicit tree of size N. In the worst case, for each node, `Floyd::sift_down` traverses down the tree from this node to a leaf. Two element comparisons are done at each level of the tree, except at the topmost level. Hence, the total number of element comparisons performed is bounded above by $\sum_{i=1}^{\lfloor N/2 \rfloor} 2h_i$. It is well-known that this sum is bounded by $2N$.

```
// Construct a heap using the repeated-merging method.
// This program is a translation of Floyd's original Algol program into C++.

#include <iterator>     // defines std::iterator_traits
#include "sift_down.cc" // defines the sift-down function

namespace Floyd {

  template <class position, class ordering>
  void make_heap(position first, position last, ordering less) {
    typedef std::iterator_traits<position>::difference_type index;
    const position a = first - 1;
    const index N = last - first;

    for (index i = N / 2; i > 0; i--)
      sift_down(a, i, N, less);
  }
}
```

Figure 3.8: Floyd's heap-construction program in C++.

When we translated Floyd's heap-construction program into pure C, we used the following principles:

1. In the sift-down function separate code was written to handle the cases when the operation proceeds down to left subtree and to the right subtree.
2. The sift-down function was inlined. Because of the above branching, two copies of the continuation code after the inner loop was created to avoid an extra unconditional jump.
3. In the connection with an index calculation, after checking the validity of the index, also the position of the node in question is computed, and the element stored at that node loaded into registers. All common subexpressions were eliminated so all these operations are carried out only once at each level.

We expect that most of these optimizations can be carried out automatically by modern compilers, so the main concern here was to get a realistic estimate for the instruction count of Floyd's program.

In the optimized version two memory reads are performed at each iteration of the inner loop, so the number of memory reads is the same as that of comparisons. Taking into account the $\lfloor N/2 \rfloor$ memory reads done in the outer loop, we get that the total number of memory reads is at most $2.5N$. At each iteration the optimized version executes exactly 6 pure-C operations in the outer loop, and at most 14 pure-C operations in the inner loop, one of which is executed before the validity test of index j causing the termination of the loop (but the inner loop can also terminate earlier). Since the two comparisons in the inner loop are performed at most N times and the outer loop is executed $\lfloor N/2 \rfloor$ times, the total number of pure-C operations executed is at most $14N + 2N + 6N/2 + O(1)$, which is $19N + O(1)$.

In Floyd's scheme two consecutive sift-down operations process two disjoint subtrees so temporal locality is absent. The poor cache behaviour is formalized in the following theorem.

Theorem 3.2 *Let B denote the size of cache blocks, M the capacity of the cache, and N the size of the heap to be constructed. Assume that initially the cache is empty and that the block-replacement strategy is LRU.*

An upper bound for the worst case. *Assuming that the cache has capacity for at least $2B$ blocks, Floyd's heap-construction program incurs never more than $2N(\log_2 B)/B + 2N/B + O(\log_2 N)$ cache misses.*

A lower bound for the worst case. *There exists an input of size $N \geq 4M$ for which Floyd's heap-construction program incurs more than $(1/2)N(\log_2 B)/B - (1/2)N/B$ cache misses.*

A lower bound for the best case. *For any input Floyd's heap-construction program will incur at least* $\max\{N/B, (3/2)N/B - M/B - 1\}$ *cache misses.*

Proof: The upper bound for the worst case. Let us first consider the nodes whose height is larger than or equal to $\log_2 B$. We make a pessimistic assumption that at every level the access of the children (which are in the same block) incurs a cache miss, and that the sift-down operations always traverse down to a leaf. It is well-known that there are at most $\lceil N/2^{h+1} \rceil$ nodes of height h in a binary tree of size N. Since the sift-down operation starting from any node of height h incurs at most $h+1$ cache misses, the total number of cache misses incurred by all the topmost nodes is bounded by

$$\sum_{h=\log_2 B}^{\lfloor \log_2 N \rfloor} \left\lceil \frac{N}{2^{h+1}} \right\rceil \cdot (h+1)\,.$$

By transforming the summation index to $i = h - \log_2 B + 1$ and using the facts that

$$\sum_{i=1}^{\infty} \frac{1}{2^i} < 1 \text{ and } \sum_{i=1}^{\infty} \frac{i}{2^i} < 2\,,$$

we get that the sum is bounded by $(N/B)\log_2 B + 2N/B + O(\log_2 N)$.

Let us next consider the nodes whose height is less than $\log_2 B$. In an implicit tree of size N the nodes of height h occupy at most $\lceil N/(B2^{h+1}) \rceil$ blocks. Assume now that the cache has capacity for at least $2B$ blocks. This means that when processing the consecutive nodes of height h, $h < \log_2 B$, that all locate in the same block, all the nodes of a smaller height possibly accessed by a sift-down operation starting from any of these B nodes can be at the same time inside the cache, since the number of blocks occupied by these nodes is at most $2B$. We make a very pessimistic assumption that, for each h, $h = 0, 1, \ldots, \log_2 B - 1$, all of the $\lceil N/B \rceil$ blocks are fetched in when processing the nodes of height h. This gives $\lceil N/B \rceil \log_2 B$ cache misses for all of the bottommost nodes. To sum up, the total number of cache misses incurred when handling all the nodes covered by these two cases is $2(N/B)\log_2 B + 2N/B + O(\log_2 N)$.

The lower bound for the worst case. Let $N \geq 4M$. For the sake of simplicity, assume that $N = 2^t - 1$ for some integer t. As a bad input we consider a tree where the node with index i contains the ith largest element. That is, if the input is seen as an indexed sequence, the elements are in ascending order. To get the lower bound, we calculate only the number of cache misses incurred when processing the nodes of height h for $h = 1, 2, \ldots, \log_2 B - 1$. The cache misses incurred for all the other nodes are excluded from our calculations.

For this particular input each sift-down operation proceeds down to a leaf. Consider now any block of size B containing nodes of height h, $1 \le h < \log_2 B$. The key observation is that the leaves visited by the sift-down operations, starting from the nodes in that block, must lie in 2^h distinct blocks. There are $\lceil N/2^{h+1} \rceil$ nodes of height h. Hence, when scanning over all the nodes of height h, $(1/2)N/B$ different blocks at the leaf level are accessed, i. e., all the blocks at the leaf level.

The input tree is assumed to be perfectly aligned, so the leaf level of the tree occupies $2^{t-1}/B$ blocks. Since $2^{t-1} \ge 2M$, none of the blocks containing the 2^{t-2} last (first) leaves can be in the cache when processing the first (last) half of nodes of height h, $1 \le h < \log_2 B$. That is, for each h, $1 \le h < \log_2 B$, all the blocks containing leaves have to be fetched into the cache at least once. Thus, the total number of cache misses is at least $(\log_2 B - 1)(1/2)N/B$ as claimed.

Proof of the lower bound for the best case. Consider any node that is the last node in its block. Clearly, there are at least N/B such nodes. Let us call the node indicated by parameter `i` of `Floyd::sift_down` the **new node**. Each sift-down operation will visit both children of the new node. Hence, when the new node scans over all branches each of N/B blocks will be accessed at least once.

Consider now any branch v that is the last node in its block. Assume that the index of v is j and that $j > M$. After the sift-down operation for v, and before that for its parent, the $B - 1$ following sift-down operations access the block containing v, but hereafter that block is accessed first when the sift-down operation for the parent of v is invocated. The $\lceil j/2 \rceil$ nodes between v and the parent of v, including these two, occupy at least $j/(2B)$ blocks. The sift-down operations here between access all these blocks and at least the same number of blocks at the level below them. So, if $j > M$, the block containing v is no more in the cache when the parent of v is considered, incurring a cache miss. This will happen for all the $(N - M - \lceil N/2 \rceil)/B$ branches whose index is larger than M. In total, the number of cache misses is at least $N/B + (N - M - \lceil N/2 \rceil)/B$, which is bounded from below by $(3/2)N/B - M/B - 1$ as claimed.

3.4.3 Layerwise construction

In the layerwise heap-construction method proposed, e. g., by Fadel et al. [FJKT99] the idea is to find the largest half of the elements, put them into the leaves of the tree, exclude the leaves from further consideration, and continue the process for the remaining nodes. Since in each round about half of the elements are excluded, the number of instructions executed is linear in the number of nodes of the input tree, if the method used for selecting the largest half of the elements is linear. Moreover, if the input tree is seen as an indexed sequence, this sequence is mainly accessed sequentially, which immediately leads to a good cache performance.

To describe the method precisely, let N be the size of the input tree and $\otimes$ the given strict weak ordering. Furthermore, let n denote the index of the last branch of the input tree. In each **round** the input is seen as an indexed sequence, the nth largest element of the sequence with respect to the *converse relation* $\otimes$ is selected, and the sequence is partitioned around the selected element e as follows: e is put to the position with index n, all elements not smaller than e to positions with smaller index, and all elements not larger than

e to positions with larger index. Since the node with index n contains e, which is not smaller than the elements at positions with index $2n$ and $2n+1$ (if valid), it can also be excluded and the process is repeated for the positions with index less than n. After at most $\lfloor \log_2 N \rfloor$ rounds, the outcome can be interpreted as a tree which clearly is an implicit maximum heap.

The efficiency of this heap-construction method depends heavily on the efficiency the selection routine used in its implementation. We would like to point out that most selection routines described in the literature are recursive in nature so they need a recursion stack for their operation, i. e., they do not run in-place, but of course a recursion stack of logarithmic size is acceptable in practice. Furthermore, both the selection and partitioning routines used should be able to handle multiset data, since the input can contain equal elements. Therefore, not all published algorithms can be used for our purposes.

The classical alternatives are the algorithm of Hoare [Hoa61], which is randomized and which executes a linear number of instructions in the average case; and the deterministic algorithm of Blum et al. [BFP$^+$73] which is based on the prune-and-search technique and which executes a linear number of instructions in the worst case. We implemented the space-efficient variant of the deterministic prune-and-search algorithm described in [HSR98, Section 3.6], but it was only a little faster than sorting (compared to `std::sort`) and several times slower than the randomized selection program. The `std::nth_element` routine of the STL (version 3.2) is a variant of Hoare's randomized scheme, but it selects the partitioning element deterministically by using the median-of-3 principle, i. e., the median of the first, the middle, and the last element. In addition to selection, `std::nth_element` carries out the partitioning around the selected element so it can readily be used in the implementation of the layerwise heap-construction method, as shown in Figure 3.9.

In order to analyse the performance of this layerwise heap-construction program, let us briefly recall the principle of Hoare's selection algorithm. Assume that we want to find the nth largest of the given N elements with respect to a given strict weak ordering $\otimes$. Now the recipe is as follows:

1. Select one of the elements as the partitioning element, denoted by y.
2. Use y to partition the input sequence into three sections

X-section	Y-section	Z-section

   such that y alone gets into the Y-section, all elements in the X-section are smaller than or equal to y, and all elements in the Z-section are larger than or equal to y.

3. Let ℓ denote the number of elements in the X-section. If $\ell \geq n$, recursively search for the nth largest element from the X-section; if y is the nth largest item, return its index as the output; otherwise, recursively search for the $(n - \ell - 1)$th largest element from the Z-section.

For the purpose of our analysis we made three changes to `std::nth_element`:

1. Instead of a deterministic selection of the partitioning element, we generate three distinct positions randomly and select the median of the elements in these positions as the partitioning element.

```
// Construct a heap layerwise.

#include <functional>           // defines std::binary_function
#include <algorithm>            // defines std::nth_element
#include "position_tied_node.cc" // defines the parent function for positions


namespace Layerwise {
  using namespace position_tied_node;
  using position_tied_node::parent;

  // The converse relation is needed since we construct a maximum heap.
  template <class ordering>
  class converse_relation
    : public std::binary_function<typename ordering::first_argument_type,
                                  typename ordering::second_argument_type,
                                  bool> {
    protected:
      ordering less;
    public:
      explicit converse_relation(const ordering& less) : less(less) {
      }

      bool operator()(const typename ordering::first_argument_type& x,
                      const typename ordering::second_argument_type& y) const {
        return less(y, x);
      }
  };

  template <class position, class ordering>
  void make_heap(position first, position beyond, ordering less) {
    converse_relation<ordering> greater = converse_relation<ordering>(less);
    const position base = first - 1;
    while (beyond - first > 1) {
      position last = beyond - 1;
      position last_branch = parent(base, last);
      std::nth_element(first, last_branch, beyond, greater);
      beyond = last_branch;
    }
  }
}
```

Figure 3.9: Layerwise heap-construction method in C++.

2. In `std::nth_element` the element value is given as the parameter for the partitioning routine, but we give the position of the partitioning element as the parameter.

3. Instead of using the normal comparison operation when comparing the elements, we let the positions of equal elements determine their order.

Especially, the last modification is important since it allows us to use earlier derived complexity results, which assume that the elements are all distinct, even though the program can handle multisets.

In Figure 3.10 an extract of this modified partitioning program is given. The three critical loops are identical in structure except for details, so only one of the loops was translated into pure C. The innermost loops are very tight including a position increment, a memory read, an element comparison, and the position comparison added by us. Further, it is seen that three pure-C instructions are executed for each element exchange. It is well-known (see, e. g., [Ram94]) that the expected number of element comparisons made by Hoare's selection routine to find the median of N elements is bounded by $3.39N$, and that selecting more extreme elements require fewer element comparisons on an average. The

```
#include <iterator>   // defines iterator_traits

template <class position, class ordering>
position unguarded_partition(position first, position pivot_position,
                             position beyond, ordering less) {

  typedef iterator_traits<position>::value_type element;
  element pivot = *pivot_position;
  position low = first - 1;
  high = beyond;

  outer_loop:
  first_while:
    ++low; left = *low;
    if (low >= pivot_position) goto loop_exit;
    if (!less(pivot, *low)) goto first_while;

  second_while:
    --high; right = *high;
    if (high <= pivot_position) goto loop_exit;
    if (!less(*high, pivot)) goto second_while;

    *low = right; *high = left;
    goto outer_loop;
  loop_exit:

  // Assert:    low == pivot_position && high > pivot_position
  //         || low < pivot_position && high == pivot_position

  if (low > pivot_position) { ... }
  else { ... }
}
```

Figure 3.10: The partition function in C++.

expected number of element exchanges performed is less than $3.39N/6$ as shown by Sedgewick [Sed77]. By inspecting the program, the number of memory reads is seen to be the same as that of element comparisons, plus the reads done in the selection of the partitioning elements which is only $O(\log_2 N)$ on an average. Clearly, the performance is better when the median-of-3 principle is used when choosing the partitioning element. By using these figures the expected number of pure-C operations executed during selection of the median is seen to be bounded by $3.39(4 + 3/6)N + O(\log_2 N) = 15.3N + O(\log_2 N)$. Further, this figure should be multiplied by 2 due to the repeated selections performed by the layerwise method. The expected pure-C cost of the layerwise method is therefore $30.6N + O(\log_2 N)$.

Fadel et al. [FJKT99] observed the goodness of the layerwise heap-construction method on a paged-memory environment. Its cache behaviour is also good, at least in theory, as shown in the next theorem.

Theorem 3.3 *Let B denote the size of cache blocks, M the capacity of the cache, and N the size of the heap to be constructed. Assume that initially the cache is empty and that the block-replacement strategy is LRU.*

An upper bound for the worst case. *It is possible to implement the layerwise heap-construction method such that it incurs never more than cN/B cache misses for a positive constant c, even if the cache can only hold a constant number of blocks.*

An upper bound for the randomized average case. *When Hoare's ran-*

domized selection scheme is used in the implementation of the layerwise heap-construction method, the expected number of cache misses incurred is no more than $7N/B + O(\log_2 N)$ if $M \geq 32B$.

A lower bound for the best case. *Any straightforward implementation of the layerwise heap-construction method will incur at least $2N/B - 2M/B - O(\log_2 N)$ cache misses.*

Proof: The upper bound for the worst case. Consider the the space-efficient deterministic selection program described in [HSR98, Section 3.6] and the 3-way partitioning program described in [Weg85]. Both of these programs access memory strictly in a sequential manner. Hence, when these programs are used as subroutines in the implementation of the layerwise heap-construction method, the resulting program will also access memory strictly in a sequential manner. Because of this the number of cache misses incurred is directly related to the number of memory references, which is known to be bounded by dN for a positive constant d. It would be easy to achieve the cache-miss bound dN/B by using the OPT block-replacement strategy, even if the capacity of its cache is $O(1)$ blocks. But, as Lemma 3.1 shows, the LRU strategy can achieve the bound $2dN/B$ if its cache is two times larger than the cache used by the OPT strategy; that is, the size of the cache is still $O(1)$ blocks.

The upper bound for the randomized average case. Let us carry out the analysis in two phases: we analyse first the compulsory cache misses incurred, if the OPT block-replacement strategy is in use, and second the slowdown caused by the LRU block-replacement strategy. Consider the sequence of memory references produced by the selection program. First, we observe that a memory write always follows a memory read accessing the same memory location. Hence, memory writes do not incur any cache misses. Second, the partitioning routine accesses memory sequentially. This means that, when the OPT block-replacement strategy is used, the cache must only fit a constant number of blocks. Actually, two blocks are seen to be enough. Thus, the number of cache misses is directly related to that of the memory reads, which in the average case is known to be bounded by $3.39N + O(\log_2 N)$.

If the cache available for the LRU strategy can fit M/B blocks, we get by Lemma 3.1 that the expected number of cache misses incurred by the selection program is at most

$$\frac{M/B}{M/B - 2 + 1} 3.39N/B + O(\log_2 N)\,.$$

When the selection program is used in the layerwise heap-construction program, this bound should be multiplied by 2. Assuming that $M \geq 32B$ the expected number of cache misses incurred is less than $7N/B + O(\log_2 N)$ as claimed.

Proof of the lower bound for the best case. By a straightforward implementation we mean one that constructs a heap one layer at a time without trying to combine the different rounds. Ignoring the cost of selection, the partitioning must read each of the elements involved. The number of elements considered in the ith round is at least $N/2^{i-1} - 1$ if the given tree stores N elements. Hence, the number of blocks accessed when constructing a heap is at least

$$\sum_{j=0}^{\ell}\left(\frac{N}{B2^i}-1\right),$$

where ℓ is an integer for which $N/2^\ell > M$, but $N/2^{\ell+1} < M$. That is, when the whole problem fits into the cache, no more cache misses are incurred. Since

$$\sum_{j=0}^{\ell}\frac{1}{2^i} = 2-\left(\frac{1}{2}\right)^\ell$$

and $\log_2(N/M) - 1 < \ell$, the above sum is bounded from below by $2N/B - 2M/B - O(\log_2 N)$.

3.5 Reengineered versions of the known methods and their analysis

In this section, we describe and analyse the reengineered versions of the methods presented in Section 3.4. Our primary intention has been to improve the cache behaviour of the methods, and our secondary intention to reduce the instruction count of the programs. As in Section 3.4, the instruction counts are measured in pure-C operations under the assumptions that the elements are one-word integers and that the given implicit tree is stored in an array.

3.5.1 Reengineered repeated insertion

Williams' original program visits the nodes of the input tree in a breadth-first manner, but this is by no means the only possibility. The precondition for the correct operation of the sift-up function is that the elements on the path from the node under consideration to the root are in ascending order with respect to the given strict weak ordering. More precisely, before the sift-up operation starting from the node with index k, for any two nodes with indices i and j, $i < j < k$, on the path from the node with index k to the root, the element stored at node with index i should not be smaller than that stored at node with index j. Given this precondition the sift-up operation will make the path one longer by extending it first with the new node and sifting the element up towards the root until the elements on the path are in ascending order again. With this description in mind, the correctness is guaranteed even if the nodes are visited in a depth-first manner. A recursive program, which constructs an implicit heap by visiting the nodes in depth-first order, is shown in Figure 3.11.

One reason for the cache inefficiency of Williams' program is the zigzag movement across the tree due to the breadth-first traversal of the nodes. The proofs of the lower-bound results in Theorem 3.1 indicate this clearly. In the next theorem we prove that the temporal locality is considerably better when the nodes are processed in a depth-first manner.

Theorem 3.4 *Let B denote the size of cache blocks, M the capacity of the cache, and N the size of the heap to be constructed. Assume that initially the cache is empty and that the block-replacement strategy is LRU.*

```
// Construct a heap using the repeated-insertion method.
// This variant visits the nodes in depth-first order.

#include <iterator>          // defines std::iterator_traits
#include "sift_up.cc"        // defines the sift-up function
#include "index_tied_node.h" // defines the index-tied node class

namespace Reengineered_Williams {
  using namespace index_tied_node;

  template<class position, class node_index, class ordering>
    void make_heap(position a, node_index i, node_index N, ordering less) {
      if (i <= N) {
        sift_up(a, i, less);
        make_heap(a, i.left_child(), N, less);
        make_heap(a, i.right_child(), N, less);
      }
    }

  template<class position, class ordering>
    void make_heap(position first, position beyond, ordering less) {
      typedef std::iterator_traits<position>::difference_type index;
      if (beyond - first < 2) return;
      make_heap(first - 1, node<index>(1), node<index>(beyond - first), less);
    }
}
```

Figure 3.11: Reengineered version of the repeated-insertion heap-construction method in C++.

An upper bound for the worst case. *Assuming that the cache can hold at least $r\lfloor\log_2 N\rfloor$ blocks for some real number $r > 1$, the reengineered version of the repeated-insertion heap-construction method incurs at most $(r/(r-1))N/B$ cache misses, which for $M/B \gg \log_2 N$ is close to N/B.*

Proof: The upper bound for the worst case. Let s be the sequence of memory references produced by the reengineered version of the repeated-insertion heap-construction method for an arbitrary input of size N. The idea in the proof is the same as that used in the proof of the upper bound for the worst case in Theorem 3.1. Namely, we present a block-replacement strategy A which guarantees that the number of cache misses incurred for sequence s, denoted by $f_A(s)$, is less than or equal to N/B if the cache can fit $m_A = \lfloor\log_2 N\rfloor + 1$ blocks. By definition $f_{OPT}(s) \le f_A(s)$, so we can use Lemma 3.1 to conclude that, for a real number $r > 1$, $f_{LRU}(s) \le (r/(r-1))N/B$ if $m_{LRU} = M/B \ge r\lfloor\log_2 N\rfloor$.

The main observation is that the reengineered version visits the nodes at any level of the initial tree from left to right. Recall that a block is called relevant to node v if it contains any node on the path from v to the root. Initially, we let the block-replacement strategy A load the blocks relevant to the left-most leaf into the cache, and thereafter keep the blocks relevant to the last-visited leaf there. All the memory accesses are performed by the sift-up operations and each sift-up operation accesses only the blocks relevant to the new node. Hence, the traversal towards the root does not cause any cache misses. However, a cache miss is possible when the new node is accessed for the first time. By substituting the block containing the new node for the block that contains nodes from the same level, the number of blocks can always be kept below $\lfloor\log_2 N\rfloor + 1$, and since the nodes at each level are visited from left to right the replaced block will not become relevant to any node later on.

For this version of the repeated-insertion heap-construction method the

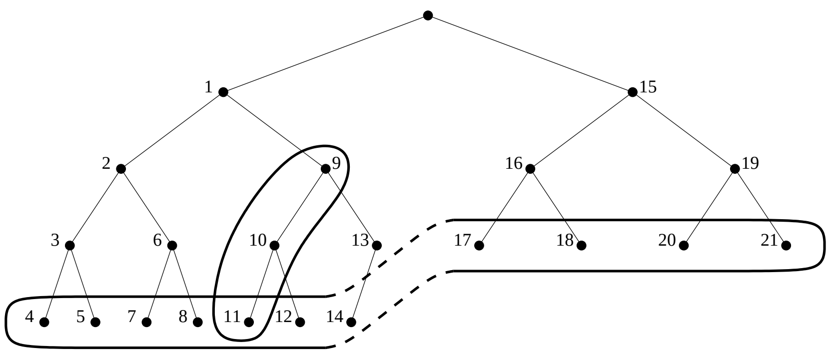

Figure 3.12: The depth-first ordering of the sift-up operations. The first loop traverses through the leaves from left to right with a break at the last leaf; the second loop traverses the tree upwards from a leaf that is a left child to the first ancestor that is a right child, performing the sift-up operation at each node on the way down; and the third loop is the inner loop in the sift-up operation itself.

memory performance is close to optimal, so its runtime performance is determined by the internal computations. Hence, it is motivated to try to reduce the number of instructions performed. In the worst case most work is done in the sift-up function. Our experiments show that for a random placement of integers $\{0, 1, \ldots, N-1\}$ into the nodes the observed number of element comparisons is close to that made by the original version. This means that the code tuning of the program should be accomplished carefully since both the inner loop of the sift-up function and the outer loop corresponding to the calls of the make-heap function are executed about equally many times. Hence, moving some code from the inner loop to the outer loop improves the worst-case performance, but the average-case performance remains the same. In addition, we wanted to modify the program so that the memory reference pattern of the tuned program is the same as that of the untuned program.

As the first step, we removed the recursion from the program of Figure 3.11. In order to avoid a recursion stack the iterative version scans through the leaves from left to right, and for each leaf it seeks for the first ancestor, possibly the leaf itself, which is a right child. Starting from this node, the elements on the path back down to the leaf are sifted up, one at a time. This way the sift-up operations are performed in the same order as those performed by the untuned program. The loops involved are further illustrated in Figure 3.12. Only the normal parent-child relationships, which are easy to calculate in an implicit tree, are needed in the realization of these loops.

We made the following improvements to the sift-up function:

1. Before inserting an element into the heap we compared it to the element stored at the root. After this the inner loop could be divided into two similar loops that both have only one termination condition: the root reached or an element larger than or equal to the inserted element found.

2. To compensate the extra element comparison, the root is kept in registers.

3. By making two copies of each of the two versions of the inner loop the

assignment `j = i` could be avoided as hinted, e.g., in [Knu98, Exercise 5.2.1.33].

In addition to these improvements specific to this heap-construction program, we applied the following general code tuning principles:

1. Multiplications and divisions by powers of 2 were replaced by shifts.
2. Common subexpressions were eliminated.
3. Function calls were inlined by hand.
4. The loops were finished conditionally as recommended by Sedgewick [Sed78].

The resulting program is given in two parts in Figures 3.13 and 3.14. Let us now analyse the worst-case number of pure-C operations executed by this program when constructing a heap of size N. During the initialization the computation of $\lfloor \log_2 N \rfloor$ requires $O(\log_2 N)$ pure-C operations. All the other work done during the initialization takes $O(1)$ pure-C operations. Next we consider the three loops separately. In the calculations below, the instructions in the continuation of a macro are counted in the place where they appear in the source code.

First, consider the loop that scans through the leaves. This loop is broken into two pieces with a break point at the last leaf. The number of leaves is $\lceil N/2 \rceil$. For every second leaf only a single sift-up operation is performed, and for the other half a sequence of multiple sift-up operations is necessary. Both in the continuation of each sift-up operation and each sequence of sift-up operations 2 pure-C operations are executed. There is an exception at the break point, where $O(1)$ extra pure-C operations are executed, but this is done only once. In total, this is $2N + O(1)$ pure-C operations.

Second, consider the work done in the macro `SIFT_UP_MULTIPLE` that takes care of a sequence of sift-up operations. Both loops in `SIFT_UP_MULTIPLE` visit each of the nodes at most once, except every second leaf. The first loop executes 3 pure-C operations and the second 2 pure-C operations at each of the $3N/4$ iterations. The macro is invoked for every second leaf meaning that the 3 pure-C operations outside the loops are executed $N/4$ times. Summing up, the number of pure-C operations executed is $4.5N + O(1)$.

Third, let us calculate the work done inside the macro `SIFT_UP`. The sift-up operation is performed at each node, except at the root. Furthermore, 5 or 7 pure-C instructions are executed before entering one of the inner loops. For a worst-case input the shorter branch is chosen at all times. This means at most $5N$ pure-C operations. Finally, both of the inner loops execute 5 pure-C operations. As the inner loop of Williams' program, the inner loops are executed at most $N \log_2 N - 1.91N + O(\log_2 N)$ times. To summarize, the tuned version of the reengineered variation of the repeated-insertion heap-construction method executes at most $O(\log_2 N) + 2N + O(1) + 4.5N + O(1) + 5N + 5(N \log_2 N - 1.91N + O(\log_2 N))$ pure-C operations, which is $5N \log_2 N + 1.95N + O(\log_2 N)$ pure-C operations.

3.5.2 Reengineered repeated merging

The main drawback in Floyd's heap-construction program is the order in which the nodes are visited. The nodes could be processed in any order, provided that

```
#define xcat(x, y) x ## y
#define cat(x, y) xcat(x, y)

#define UP_TO_ROOT(p, current, continuation)                            \
  cat(up_to_root, cat(current, __LINE__)):                              \
    i = i.parent(); p = base + i;                                       \
    temp = *p; *current = temp;                                         \
    if (i > 1) goto cat(up_to_root, cat(p, __LINE__));                  \
    continuation

#define UP_TO_SENTINEL(p, current, hole, continuation)                  \
  cat(up_to_sentinel, cat(current, __LINE__)):                          \
    *hole = temp;                                                       \
    i = i.parent(); p = base + i;                                       \
    temp = *p;                                                          \
    if (less(temp, in)) goto cat(up_to_sentinel, cat(p, __LINE__));     \
    continuation

#define SIFT_UP(new_node, continuation) {                               \
    position p, q;                                                      \
    element in, temp;                                                   \
    node<index> i;                                                      \
    q = base + new_node;                                                \
    in = *q;                                                            \
    if (less(top, in)) goto cat(up_to_root, cat(new_node, __LINE__));   \
    i = new_node.parent(); p = base + i;                                \
    temp = *p;                                                          \
    if (less(temp, in)) goto cat(up_to_sentinel, cat(new_node, __LINE__));\
    continuation;                                                       \
  cat(up_to_sentinel, cat(new_node, __LINE__)):                         \
    UP_TO_SENTINEL(q, p, q, *p = in; continuation);                     \
    UP_TO_SENTINEL(p, q, p, *q = in; continuation);                     \
  cat(up_to_root, cat(new_node, __LINE__)):                             \
    i = new_node;                                                       \
    top = in;                                                           \
    UP_TO_ROOT(p, q, *p = in; continuation);                            \
    UP_TO_ROOT(q, p, *q = in; continuation);                            \
  }

#define SIFT_UP_MULTIPLE(leaf, continuation) {                          \
    node<index> j = leaf;                                               \
    do { j = j.parent(); } while (j.is_left_child());                   \
    if (j == 1) j = 2;                                                  \
  cat(sift_up_multiple, __LINE__):                                      \
    SIFT_UP(j,                                                          \
            j = j.left_child();                                         \
            if (j <= leaf) goto cat(sift_up_multiple, __LINE__);        \
            continuation);                                              \
  }
```

Figure 3.13: Macros for sifting up one or more elements.

```
// Construct a heap using the repeated-insertion method.
// This variant visits the nodes in depth-first order.

#include <iterator>          // defines std::iterator_traits
#include "log_2.cc"          // defines the log_2 function
#include "index_tied_node.h" // defines the index-tied node class
#include "sift_up.h"         // defines macros for the sift-up operation

namespace Tuned_Reengineered_Williams {
  using namespace index_tied_node;

  template<class position, class ordering>
  void make_heap(position first, position beyond, ordering less) {
    typedef std::iterator_traits<position>::value_type element;
    typedef std::iterator_traits<position>::difference_type index;

    const node<index> N = beyond - first;
    const position base = first - 1;
    element top = *first;
    const node<index> leftmost_leaf = 1 << log_2(N);
    node<index> leaf = leftmost_leaf;

    // Scan the leaves from left to right, and use insertion-sort to sort
    // the elements on the path to the root from each leaf.

  left2right_loop1:
    SIFT_UP_MULTIPLE(leaf, goto left2right1);
    left2right1:
    ++leaf;
    SIFT_UP(leaf,
            ++leaf;
            if (leaf < N) goto left2right_loop1;
            if (leaf == N) goto last_leaf1;
            leaf = N.parent().successor();
            if (leaf.is_right_child()) goto last_leaf2;
            if (leaf < leftmost_leaf) goto left2right_loop2;
            return);
  last_leaf1:
    SIFT_UP_MULTIPLE(leaf,
                     leaf = N.parent().successor();
                     if (leaf.is_right_child()) goto last_leaf2;
                     if (leaf < leftmost_leaf) goto left2right_loop2;
                     return);
  last_leaf2:
    SIFT_UP(leaf,
            ++leaf;
            if (leaf < leftmost_leaf) goto left2right_loop2;
            return);
  left2right_loop2:
    SIFT_UP_MULTIPLE(leaf, goto left2right2);
  left2right2:
    ++leaf;
    SIFT_UP(leaf,
            ++leaf;
            if (leaf < leftmost_leaf) goto left2right_loop2;
            return)
    }
}
```

Figure 3.14: Tuned version of the reengineered repeated-insertion heap-construction method in C++.

```
// Construct a heap using the repeated-merging method.
// This variant visits the nodes in depth-first order.

#include <iterator>     // defines std::iterator_traits
#include "sift_down.cc" // defines the sift-down function

namespace Reengineered_Floyd {

  template <class position, class node_index, class ordering>
    void make_heap(const position base, node_index root, const node_index last,
                   ordering less) {
      const node_index last_branch = last.parent();
      node_index child = root.right_child();
      if (child <= last_branch)
        make_heap(base, child, last, less);
      child = child.left_sibling();
      if (child <= last_branch)
        make_heap(base, child, last, less);
      sift_down(base, root, last, less);
    }

  template <class position, class ordering>
    void make_heap(position first, position beyond, ordering less) {
      typedef iterator_traits<position>::difference_type index;

      const position base = first - 1;
      if (beyond - first < 2) return;
      make_heap(base, node<index>(1), node<index>(beyond - first), less);
    }
}
```

Figure 3.15: Reengineered version of the repeated-merging heap-construction method in C++.

the subtrees of a node form a heap before the node is considered as a root of the larger subtree. Actually, temporal locality can be improved considerably by processing, e. g., the right subtree of a node, then its left subtree, and directly after this the subtree rooted by the node itself. A simple recursive C++ program implementing this idea is given in Figure 3.15.

In the next theorem we prove that the cache behaviour of the reengineered program is superior to Floyd's original proposal.

Theorem 3.5 *Let B denote the size of cache blocks, M the capacity of the cache, and N the size of the heap to be constructed. Assume that initially the cache is empty and that the block-replacement strategy is LRU.*

An upper bound for the worst case. *The reengineered version of the repeated-merging heap-construction method incurs never more than $N/B + NB(\log_2(M/B))/M + 2NB/M + O\left((\log_2 N)^2 + \log_2 N \cdot \log_2(M/B)\right)$ cache misses, which is close to N/B if $M/B \gg \log_2(M/B)$.*

Proof: The upper bound for the worst case. Assume that $N > M$, since otherwise the number of cache misses incurred cannot be larger than N/B. Consider first all the nodes whose height is exactly $\log_2(M/B) - 1$. Clearly, all the $2^{\log_2(M/B)} - 1 = M/B - 1$ descendants of such a node can occupy at most $M/B - 1$ blocks, so all these blocks will fit in the cache simultaneously. Hence, we have to touch the blocks that occupy the nodes of height $\log_2(M/B) - 1$ or less at most once when performing the sift-down operations for these nodes. Therefore, the total number of cache misses incurred when handling the nodes of height $\log_2(M/B) - 1$ and less is at most N/B.

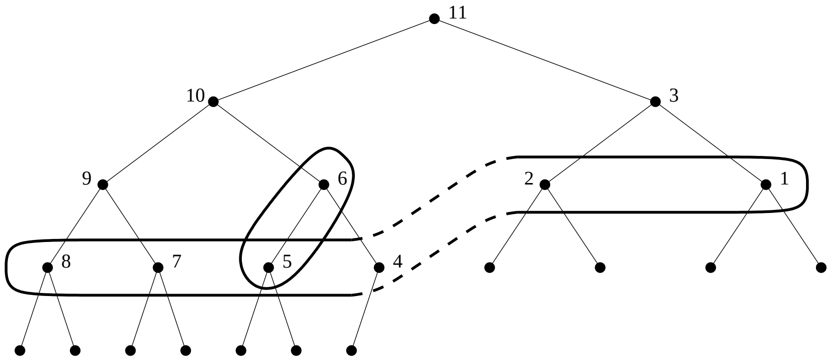

Figure 3.16: The depth-first ordering of the sift-down operations. The first loop traverses through the branches of height one from right to left with a break at the last branch; the second loop traverses the tree upwards from a branch that is a left child to the first ancestor that is a right child, performing the sift-down operation at each branch on the way up; and the third loop is the inner loop in the sift-down operation itself.

Consider now the nodes whose height is larger than $\log_2(M/B) - 1$. The worst-case scenario when invoking the sift-down function for any of these nodes is that the sift-down operation will propagate all the way to a leaf and cause one miss at each level visited. Since the number of nodes of height h is bounded by $\lceil N/2^{h+1} \rceil$, we get that the number of cache misses incurred when handling the topmost nodes is at most

$$\sum_{h=\log_2(M/B)}^{\lfloor \log_2 N \rfloor} \left\lceil \frac{N}{2^{h+1}} \right\rceil \cdot (h+1) \, .$$

This sum is bounded by

$$(NB \log_2(M/B))/M + 2NB/M + O\left((\log_2 N)^2 + \log_2 N \cdot \log_2(M/B)\right) \, .$$

This together with N/B cache misses calculated for the bottommost nodes gives the claimed bound.

Due to the recursion stack the recursive program does not construct a heap in-place, but as for the reengineered version of Williams' program the recursion can be removed such that only a constant amount of indices and positions are in use. Now the iterative version scans the nodes of height one, i. e. the bottommost branches, from right to left. For each such branch it first makes its subtree a heap by a call to the sift-down function, and thereafter it does sift-down operations along the path towards the root until a sift-down operation is done for a node which a right child. Figure 3.16 illustrates the order of processing the nodes, as well as the loops involved.

To make the program comparable with the reengineered version of Williams' program, we tuned it by using the same general principles as described in the previous section with some additions specific to the repeated-merging method.

1. In the sift-down function separate code was written to handle the case

when the first child, respectively the second child, is the larger to avoid the assignment `j = j + 1`.

2. The inner loop of the sift-down function uses the position-tied representation of the nodes to avoid index calculations.

3. Three copies of the inner loop was created in order to recall the position of a node from the previous iteration. In each copy of the loop one of the three position variables takes on the role as the parent position, thereby avoiding the assignment which corresponds to the index assignment `i = j` in the original program.

4. The test of the special case, whether a node has only one child, was moved outside the inner loop. Due to this change the sift-down function is no longer able to handle a subtree with only two nodes. A separate macro to handle this case was written. Note that only one such subtree can exist.

5. A specialized version of the sift-down function was written which handles a tree of exactly three elements. Since roughly half of the calls to the sift-down function are handled by this specialized macro, noticeable savings are achieved.

6. A specialized template function was written which computes the left child of a node when its position is an ordinary pointer. A cast was necessary to avoid the extra multiplications and divisions that were generated by our compiler when adding a difference of two positions to a position. These details could be elegantly hidden into the specialized template function as shown below:

```
template<class position>
inline position left_child(position base, position node) {
  return node + (node - base);
}

// Specialization for pointers
template<class element>
inline element* left_child(element* base, element* node) {
  return (element*)((char*) node + ((char*) node - (char*) base));
}
```

The resulting macro definitions for the sift-down operation are given in Figure 3.17, and the tuned version of the repeated-merging heap-construction method in Figure 3.18. To help understanding the tuned sift-down macro in Figure 3.17 it should be noted that each of the three instances of the `LEVEL_DOWN` macro implements one copy of the sift-down loop. The loop control of the sift-down loop is divided between the `NEW_PARENT` macro and `LEVEL_DOWN`. The loop may terminate either because the parent is found to be larger than each of the children (handled by `LEVEL_DOWN`), or it may terminate because the bottom of the tree has been reached (handled by `NEW_PARENT`). The two functions, `SIFT_DOWN3` and `SIFT_DOWN23`, are not listed. These functions implement the sift-down operation for a tree with exactly three elements and for a tree with two or three elements, respectively.

When analysing the pure-C cost we see that the sift-down specialized for three elements is called at least $\lfloor N/4 \rfloor - 1$ times and at most $\lfloor N/4 \rfloor$ times, and it executes at most 10 pure-C operations. So the `SIFT_DOWN3` macro contributes at most $10N/4$ pure-C operations. It is known that Floyd's method

```
#define xcat(x, y) x ## y
#define cat(x, y) xcat(x, y)

#define NEW_PARENT(parent, child1, continuation)                   \
    child1 = left_child(base, parent);                             \
    if (child1 < last) goto cat(cat(lvl_dwn, parent), __LINE__);   \
    if (child1 == last) {                                          \
      value1 = *child1;                                            \
      if (less(in, value1)) {                                      \
        *parent = value1; *child1 = in; continuation;              \
      } else {                                                     \
        *parent = in; continuation;                                \
      }                                                            \
    } else {                                                       \
      *parent = in; continuation;                                  \
    }

#define LEVEL_DOWN(parent, child1, child2, continuation)           \
  cat(cat(lvl_dwn, parent), __LINE__):                             \
    child2 = child1 + 1;                                           \
    value1 = *child1; value2 = *child2;                            \
    if (less(value1, value2))                                      \
      if (less(in, value2)) {                                      \
        *parent = value2;                                          \
        NEW_PARENT(child2, parent, continuation);                  \
      } else {                                                     \
        *parent = in; continuation;                                \
      }                                                            \
    else                                                           \
      if (less(in, value1)) {                                      \
        *parent = value1;                                          \
        NEW_PARENT(child1, child2, continuation);                  \
      } else {                                                     \
        *parent = in; continuation;                                \
      }

#define SIFT_DOWN(top, continuation)                               \
{ element in, value1, value2;                                      \
  position ptr1, ptr2, ptr3;                                       \
  ptr1 = top;                                                      \
  in = *ptr1;                                                      \
  ptr2 = left_child(base, ptr1);                                   \
  LEVEL_DOWN(ptr1, ptr2, ptr3, continuation);                      \
  LEVEL_DOWN(ptr2, ptr3, ptr1, continuation);                      \
  LEVEL_DOWN(ptr3, ptr1, ptr2, continuation);                      \
}
```

Figure 3.17: Definition of the sift-down operation as a macro.

```
#include <iterator>              // defines std::iterator_traits
#include "log_2.cc"              // defines the log_2 function
#include "index_tied_node.h"     // defines the index-tied node class
#include "position_tied_node.cc" // defines the parent and left-child functions for positions
#include "sift_down.h"           // defines the sift-down operation as a macro
#include "sift_down23.h"         // defines the sift-down macro for the nodes of height one

namespace Tuned_Reengineered_Floyd {
  using namespace position_tied_node;

  #define DO_PARENTS(child, continuation)                              \
  if (child.is_right_child()) {                                        \
    continuation;                                                      \
  } else {                                                             \
    cat(do_parents, __LINE__):                                         \
      child = child.parent();                                          \
      SIFT_DOWN(base + child,                                          \
                if (child.is_left_child()) goto cat(do_parents, __LINE__);\
                continuation);                                         \
  }

  #define RIGHT2LEFT(right, left, j, continuation)                     \
    cat(r2l_loop, __LINE__):                                           \
      SIFT_DOWN3(base + right, goto cat(inner_r2l_init, __LINE__));    \
    cat(inner_r2l_init, __LINE__):                                     \
      j = right;                                                       \
      DO_PARENTS(j,                                                    \
                 --right;                                              \
                 if (right >= left) goto cat(r2l_loop, __LINE__);      \
                 continuation)

  template <class position, class ordering>
  void make_heap(position first, position beyond, ordering less) {
    typedef std::iterator_traits<position>::difference_type index;
    typedef std::iterator_traits<position>::value_type element;
    typedef index_tied_node::node<index> node_index;

    node_index i, j;
    const node_index N = beyond - first;
    const node_index last_height1 = N.parent();
    const node_index first_height1 = last_height1.parent().successor();
    node_index leftmost_height1;
    position base = first - 1;
    position top;
    position last = beyond - 1;

    if (N < 2) return;
    leftmost_height1 = 1 << log_2(last_height1);

    if (first_height1 == leftmost_height1) goto last_branch;
    i = leftmost_height1.predecessor();
    RIGHT2LEFT(i, first_height1, j, goto last_branch);

  last_branch:
    top = base + last_height1;
    sift_down23(base, top, last, less);
    j = last_height1;
    DO_PARENTS(j, goto left_branches);

  left_branches:
    if (last_height1 <= leftmost_height1) return;
    i = last_height1.predecessor();
    RIGHT2LEFT(i, leftmost_height1, j, return);
  }
}
```

Figure 3.18: Tuned version of the reengineered repeated-merging heap-construction method in C++.

constructs a heap of size N with at most $2N$ element comparisons. Of these at least $N/2 - O(1)$ are performed by the specialized `SIFT_DOWN3` macro and the remaining $(3/2)N + O(1)$ by the normal `SIFT_DOWN` macro. Since 2 comparisons are performed in each iteration of the inner loop of the general sift-down, we conclude that it it iterates $(3/4)N + O(1)$ times in the worst case. All branches of the three copies of the inner loop execute 9 pure-C operations and upon inner loop exit at most 2 additional pure-C operations are executed. Furthermore, 4 pure-C operations are executed in preparation of the inner loop in the `SIFT_DOWN` macro. Hence, the general sift-down macro will execute at most $9(3/4)N + (2+4)(N/4)$ pure-C operations. The two outer loops in the `RIGHT2LEFT` macro iterate $N/4$ times in total and execute 5 pure-C operations per iteration. Also, the middle loop in the `DO_PARENTS` macro iterates $N/4$ times and it consumes 6 pure-C units per iteration. In the initialization the computation of $\lfloor \log_2 N \rfloor$ takes $O(\log_2 N)$ time. To sum up, we have a total pure-C cost of $10N/4 + 9(3/4)N + 6N/4 + 5N/4 + 6N/4 + O(\log_2 N) = 13.5N + O(\log_2 N)$.

3.5.3 Reengineered layerwise construction

There are two natural ways of improving the cache behaviour of the layerwise heap-construction method:

1. to find a better selection and partitioning method, and/or
2. to carry out several selection and partitioning rounds in one scan.

Both of these approaches has been discussed by Sibeyn [Sib99] in an external-memory setting. We have followed the first alternative by implementing the random-sampling selection method of Floyd and Rivest [FR75] in the form proposed by Sibeyn [Sib99]. However, we decided not to try the second alternative since we wanted to keep the instruction count of our program low. In particular, we have not made any attempts to improve the worst-case performance of the layerwise heap-construction method.

Let us briefly recall the random-sampling selection algorithm for selecting the nth largest of the given N elements with respect to a given strict weak ordering $\otimes$. For the time being assume that all the elements are distinct. The **rank** of an **element** in a sequence is the number of elements in the sequence that are no larger than the element itself. Now the selection process is as follows:

1. If N is relatively small, solve the problem by using any method that is known fast for small inputs. The threshold value is to be determined experimentally.
2. Fix parameters S and Δ, S being the size of the sample and Δ the safety margin. As recommended by Sibeyn, we have used the values $S = \log_2^{1/3} N \cdot (N/B)^{2/3}$ and $\Delta = (\log_2 N \cdot S)^{1/2}$.
3. Randomly and uniformly select a sample of size S out of the N input elements.
4. Use any selection method, e.g., `std::nth_element`, to find the two elements x_{low} and x_{high} whose rank in the sample is $\lfloor S/N \cdot n \rfloor - \Delta/2$ and $\lfloor S/N \cdot n \rfloor + \Delta/2$, respectively.

5. Partition the whole input into three subsequences

X-section	Y-section	Z-section

   such that all elements in the X-section are smaller than x_{low}, all elements in the Y-section larger than or equal to x_{low} but smaller than or equal to x_{high}, and all elements in the Z-section larger than x_{high}.

6. Let ℓ denote the size of the X-section after the partitioning, and m that of the Y-section. If $n \leq \ell$, use any known method to partition the X-section around its ℓth largest element. If $\ell < n < \ell + m$, do the partitioning for the Y-section around its $(n - \ell)$th largest element. Otherwise, if $n \geq \ell + m$, do the partitioning for the Z-section around its $(n - \ell - m)$th largest element.

In order to adopt this method for multisets, we compare the elements lexicographically. If two elements are equal, their positions (or indices) are used to determine their order making them all distinct. Especially, in Step 3 we use extra space to gather the elements of the sample together, and in connection with each element we also store its original position. Then in Step 4 we call `std::nth_element` twice to select the required elements, and use the stored positions in comparisons. This guarantees that the Y-section will be small with high probability with respect to the original lexicographic ordering. In Steps 5 and 6 the actual positions of the elements are used in comparisons, so no extra space is required. Normally, the computational costs are dominated by that of Step 5 since the amount of data in other steps is expected to be small. So the basic difference when handling multisets is the lexicographic comparisons performed in 3-way partitioning; in the case of distinct elements normal comparisons would have been enough.

The partitioning scheme used by us is similar to that used by Wegner [Weg85] to handle equal elements in quicksort, except that we partition the input around two partitioning elements instead of one. We maintain the loop invariant

Y-section	X-section	$?_1$	$?_2$	Z-section	Y-section

where $?_1$ and $?_2$ indicate the elements under consideration in the unexplored section. If $?_1$ (resp. $?_2$) belongs to the X-section (resp. Z-section), it is simply skipped making the X-section (resp. Z-section) one larger. If $?_1$ belongs to the Z-section, we must either exchange $?_1$ with $?_2$, if $?_2$ belongs to the X-section, or otherwise we must make a three element rotation involving $?_1$, $?_2$ and the first element in the X-section. If $?_1$ belongs to the Y-section, we must either exchange $?_1$ with the first element in the X-section and $?_2$ with the last element in the Z-section, if $?_2$ belongs to the Y-section, or otherwise we must perform a rotation of $?_1$, $?_2$ and the first element in the Z-section. This process is continued until the whole sequence is partitioned. After partitioning, the leftmost and rightmost Y-sections are brought to the middle by performing two block swaps. Since the expected number of elements in the Y-section is small, this is faster than the method based on the perhaps more natural loop invariant

X-section	Y-section	$?_1$	$?_2$	Z-section

since this will move the elements of the Y-section $O(N)$ times.

Sibeyn [Sib99] proved that with high probability the paging performance of the random-sampling selection routine is close to optimal. The same is true for the cache performance even if the LRU strategy is used in the block replacements since the sequence is mainly accessed sequentially. We rely on random access only in sampling, but since the size of the sample is relatively small, the cost caused by these accesses is vanishing. Sibeyn's calculations give directly an upper bound for the cache behaviour if the OPT strategy were used in block replacements. Let us now analyse the cache performance under the LRU block-replacement strategy.

Consider the sequence of memory references produced by the program. First, we observe that a memory write always follows a memory read accessing the same memory location. Hence, memory writes do not incur any cache misses. Second, all the subroutines used access the memory sequentially. This means that, when the OPT block-replacement strategy is used, the cache needs only to fit a constant number of cache blocks. Actually, four blocks are seen to be enough. Third, the number of cache misses is directly related to that of the memory reads. In the normal case that number is $N/B+O(N^{2/3}(\log_2 N \cdot B)^{1/3})$, but in the case of failure that number is $4.39N/B + O(N^{2/3}(\log_2 N \cdot B)^{1/3})$ in which the leading term is caused by the 3-way partitioning and a possibly a large final selection. However, Sibeyn [Sib99] proved that the failure probability is less than $1/N^{\varepsilon}$ for a positive $\varepsilon \geq 1/6$. So the expected number of cache missed incurred when using the OPT block-replacement strategy is simply the sum of the cost of the subcases times their probabilities:

$$\frac{N}{B} + \frac{1}{N^{1/6}} \cdot 4.39\frac{N}{B} + O\left(N^{2/3}(\log_2 N \cdot B)^{1/3}\right) .$$

By Lemma 3.1, if the cache used by the LRU strategy fits M/B blocks, the expected number of cache misses incurred is that bound multiplied by $(M/B)/((M/B) - 4 + 1)$.

The cache behaviour of the selection method can directly be used in the analysis of the cache behaviour of the layerwise heap-construction method.

Theorem 3.6 *Let B denote the size of cache blocks, M the capacity of the cache, and N the size of the heap to be constructed. Assume that initially the cache is empty and that the block-replacement strategy is LRU.*

An upper bound for the randomized average case. *It is possible to implement the layerwise heap-construction method such that it incurs less than $4.63N/B + O(N^{2/3}(\log_2 N \cdot B)^{1/3})$ cache misses on an average if $M > \max\{32B, 2^{12}\}$.*

Proof: The upper bound for the randomized average case. In the layerwise heap-construction method the size of the sequences considered by the selection method reduces geometrically. When the problem gets smaller, the probability of the success for the random-sampling selection method gets smaller, too. In the ith round, the problem size is bounded from below by $N/2^{i-1} - 1$ and above by $\lceil N/2^{i-1} \rceil$. When the problem gets smaller than M, it fits into the cache and no more cache misses are incurred. Hence, assuming that $M > 2^{12}$, the failure probability is never larger than $1/4$. Hence, we have the following upper bound for the expected number of cache misses incurred:

$$\frac{M/B}{M/B - 3} \sum_{i=0}^{\ell} \left(\frac{N}{B2^i} + \frac{1}{4} \cdot \frac{4.39N}{B2^i} \right) + O(N^{2/3}(\log_2 N \cdot B)^{1/3})$$

where $\ell \leq \log_2(N/M)$. Using the fact that $\sum_{i=0}^{\ell} 1/2^i < 2$ and assuming that $M \geq 32B$, the claimed bound is obtained.

3.6 Test results

The experimental part of this work involved four activities. First, a simple trace-driven cache simulator was build. Second, above this simulator a visualization tool was build which could show the memory reference patterns graphically. Both of these tools are described in detail in the M. Sc. thesis of the third author [Spo99]. Many of the findings in the theoretical part evolved first after we got a deep understanding of the programs with these tools. Third, the micro benchmark described in Section 3.2.4 was developed and this was used to extract information about the memory latencies of our computers. Fourth, the actual tests with the programs described in this paper were carried out. In this section we report the results of these experiments.

Based on the micro benchmarking we selected two computers for our tests: IBM RS/6000 397 and IBM 332 MHz F50. In brief, these computers could be characterized as follows: the former has a slow processing unit and fast memory, whereas the latter has a fast processing unit and slow memory. For further characteristics of these computers we refer back to Table 3.1. The execution trials were carried out on dedicated hardware, so no cache interference from other sources was expected.

All the programs described in Sections 3.4 and 3.5 as well as the heap-construction program from the STL (version 3.2) were included in our tests. The programs for the layerwise heap-construction method were optimized for inputs with distinct elements. The programs developed were compiled by the Kuck & Associates `KAI C++` compiler (version 3.2f), and the options used were `+K3 -O3 -qtune=pwr2 -qarch=pwr2` on IBM RS/6000 and `+K3 -O3 -qtune=604 -qarch=ppc` on IBM F50. We used several other C++ compilers to compile the programs, but none of these were able to generate as efficient code as Kuck & Associates `KAI C++` compiler. Specially, some compilers were directly hostile for our generic assembler programming.

In the experiments the input was an array of N integers, each 4 bytes or 32 bits long. The problem size N ranged from 10 to $10 \cdot 2^{20}$. For each value of N, the experiment was repeated $\lfloor 2^{24}/N \rfloor$ times, each time with a new array. Three different kinds of input data were used: an ascending sequence $\{0, 1, \ldots, N-1\}$, a descending sequence $\{N-1 \ldots, 1, 0\}$, and a random permutation of $\{0, 1, \ldots, N-1\}$. Permutations were generated by using the STL routine `std::random_shuffle()`. These data sets were selected since an ascending sequence maximizes and a descending sequence minimizes the number of comparisons performed by Williams' and Floyd's programs.

The execution times on the two test computers are given in Figures 3.19 and 3.20 (ascending input), 3.21 and 3.22 (descending input), and 3.23 and 3.24 (random input). In each graph the size of each cache level is indicated by a vertical line.

The results on IBM RS/6000 indicate clearly the effectiveness of our code tuning. The analytical pure-C costs reflects reasonably well the speed-ups obtained, though the pure-C model tends to overestimate the value of code tuning slightly. On the other hand, on a computer with a slow memory the performance of a program cannot be improved much by the traditional code-tuning techniques.

The results on IBM F50 show the effectiveness of our memory tuning. In particular, the reengineered version of the repeated-merging method performs

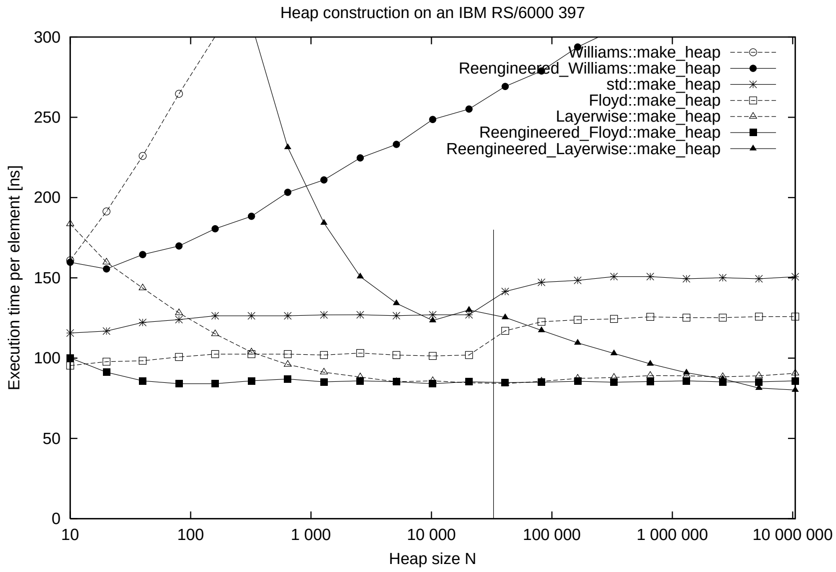

Figure 3.19: Execution times of various heap-construction programs on an IBM RS/6000 for an ascending sequence $\{0, 1, \ldots, N-1\}$.

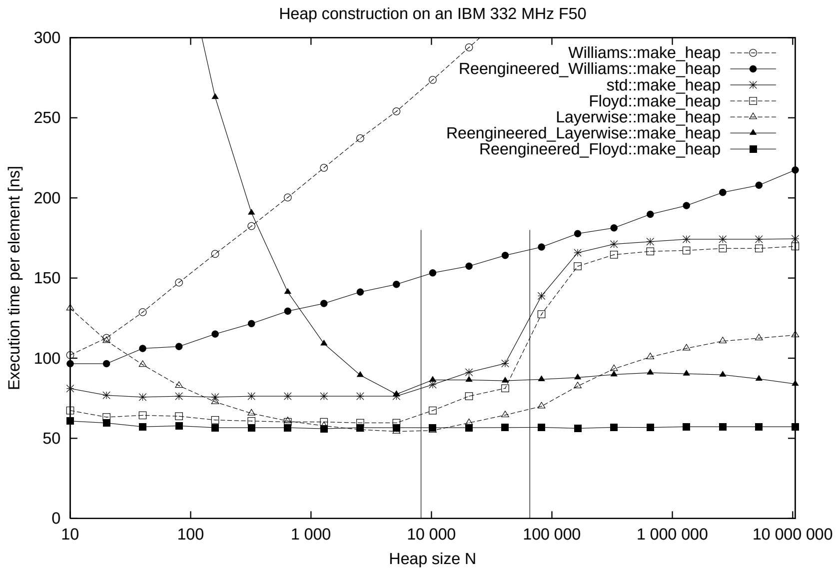

Figure 3.20: Execution times of various heap-construction programs on an IBM F50 for an ascending sequence $\{0, 1, \ldots, N-1\}$.

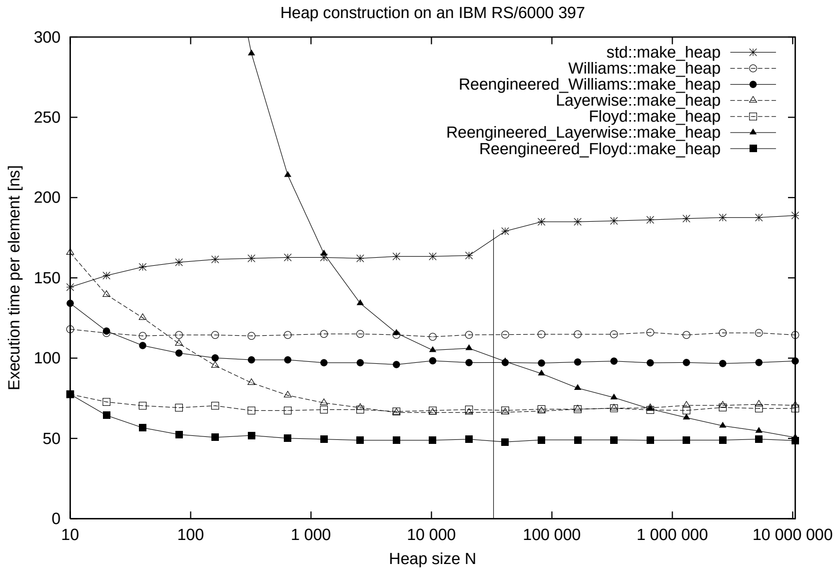

Figure 3.21: Execution times of various heap-construction programs on an IBM RS/6000 for a descending sequence $\{N-1, N-2, \ldots, 1, 0\}$.

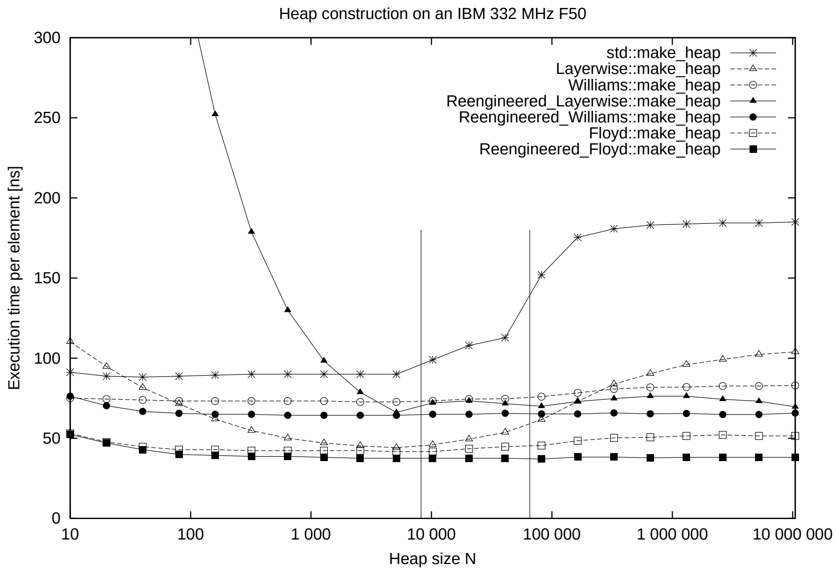

Figure 3.22: Execution times of various heap-construction programs on an IBM F50 for a descending sequence $\{N-1, N-2, \ldots, 1, 0\}$.

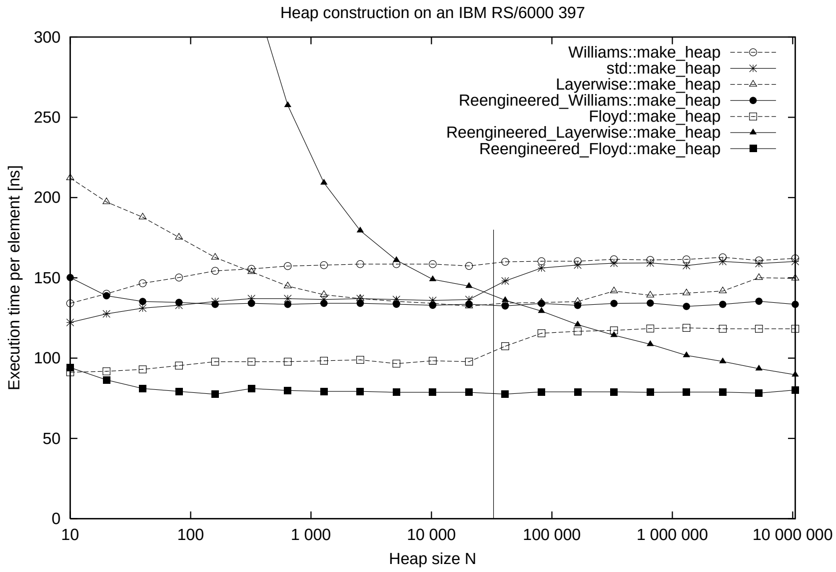

Figure 3.23: Execution times of various heap-construction programs on an IBM RS/6000 for a random permutation of $\{0, 1, \ldots, N-1\}$.

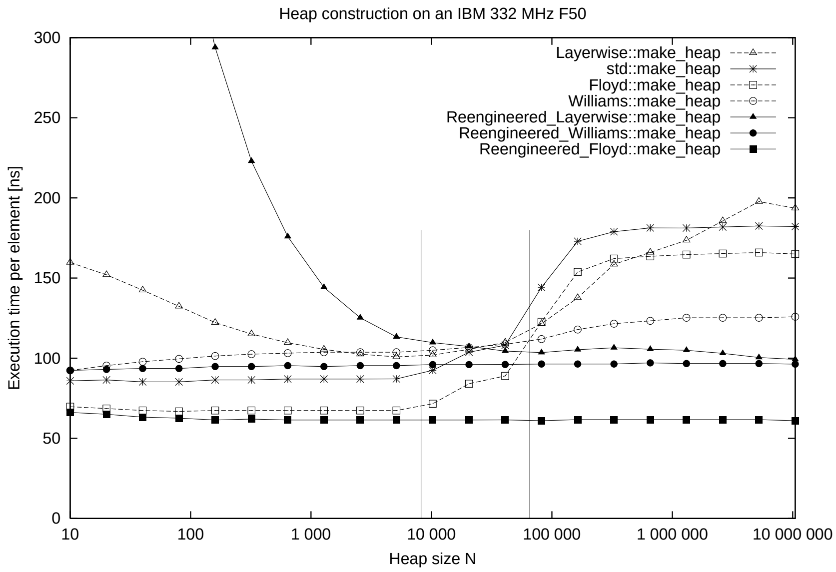

Figure 3.24: Execution times of various heap-construction programs on an IBM F50 for a random permutation of $\{0, 1, \ldots, N-1\}$.

well independently of the problem size. In the runtime curves of the other programs, especially in that of Floyd's original program, two hops can be recognized. These corresponds to the points where the size of on-chip cache and off-chip cache is exceeded. This kind of slowdown, when the size of a problem increases, is a characteristic of programs that are not designed for a computer with a hierarchical memory.

On the IBM RS/6000 (see Figure 3.19) Williams' program behaves badly compared to other methods due to its $O(N \log_2 N)$ worst-case instruction count, whereas on the IBM F50 (see Figure 3.20) it manages relatively well due to its good memory performance. So if the cache-miss rate is considerably lower than the instruction rate, instruction count is still an important measure of performance. On the IBM F50 for a random input Williams' program is faster than Floyd's program. This supports the earlier experiments carried out by Lamarca and Ladner [LL96].

The relatively good behaviour of the layerwise heap-construction method, at least for large inputs, was a surprise for us since usually methods relying on selection are considered to be slow in practice. However, we also experimented with programs, which were better at handling multisets following the ideas in the theoretical parts of Sections 3.4.3 and 3.5.3, but our preliminary implementations were not competitive. At the moment we are willing to follow Sedgewick's recommendation [Sed77] not to add any overhead to deal with equal elements.

3.7 Concluding remarks

The problem of constructing a heap is well studied and well understood. The main goal in this paper has been to understand the execution environment where the heap-construction programs are run. Most modern computers have a hierarchical memory whereas many classical algorithms, on which the current software is based, are designed for a one-dimensional flat memory. We showed that by a careful memory tuning, without forgetting the traditional code tuning, the performance of heap-construction programs could be improved by a factor of two to three.

Divide-and-conquer algorithms lead to fast programs when run on a computer with a hierarchical memory. Both our reengineered versions of the repeated-insertion and repeated-merging heap-construction methods can be seen as an incarnation of the divide-and-conquer technique. The basic idea is to traverse a tree in depth-first order, instead of breadth-first or breadth-last order as proposed originally. This makes the memory references local and improves the cache behaviour considerably. As compared to Williams' and Floyd's programs, our reengineered versions reduce the number of cache misses incurred by about a factor of 2 and $\log_2 B$, respectively.

When analysing the cache behaviour of the programs, the cache simulator built turned out to be a valuable tool. There are many similar simulators available, but according to our experience most of them are hardware-dependent. Our own tool is primitive and too slow for production use. Therefore, a fast general-purpose cache simulator is still in our desire list.

Code tuning has also played an important role in our work. Our main inspiration came from Knuth's books [Knu97, Knu98]. We hope that we could present many of the old techniques in an accessible form. Code tuning is tedious and error-prone so good software tools are needed to make it easier and safer. For instance, it would be nice to have a compiler that translates, e. g., C++ into pure C, in addition to a vendor-dependent assembler language. It would also be natural to build a profiler above the compiler which computes the pure-C cost of a program for some particular input.

When developing our C++ programs, it became clear for us that high-quality C++ compilers are hard to find. All compilers we tried gave at some point an error message "internal compiler error". The reason was often, but not always, our syntactically incorrect program, but these syntax errors were difficult to find since not even the compiler could help us. A stable compiler is a must in all software development.

One could ask how sensitive are the theoretical results derived on the assumptions made. Firstly, in current computers the replacement of the cache blocks is not realized by using the least-recently-used strategy, and the caches are not fully associative. However, as our experiments indicate, there is a good correspondence between the theory developed and practice. The replacement strategies actually in use seem to approximate reasonably well the least-recently-used principle which is known to perform well even when compared to the optimal replacement strategy, as shown by Sleator and Tarjan [ST85]. In spite of this one should be aware of the possible cache interference problems. The technology is also changing all the time. For example, there already exists cache systems that switch to the most-recently-used replacement strategy when it observes that the memory is accessed sequentially.

Secondly, a perfect alignment was assumed to simplify the proofs of the theorems, but even without this the heap-construction programs perform well. One can see an effect in the lower order terms in the functions expressing the cache performance. However, in the implementation of multiway heaps alignment is an important issue as reported by LaMarca and Ladner [LL96].

Thirdly, the assumption that $B \geq 2$ is essential for our results even if it looks innocent. If we were manipulating elements that are larger than the blocks, the number of memory reads is a relevant goodness measure for heap-construction programs. If the input elements are large, moves are expensive. In this case temporal locality is more important than spatial locality. We tested our programs for double-precision floating-point numbers, but basically the results were the same as those reported for unsigned integers. However, already then it was possible to observe a small slowdown in the performance of the repeated-insertion and layerwise heap-construction methods which is mainly due to the fact that the number of memory reads performed is higher for these methods than that for the other methods.

Finally, we would like to point out that current computers have many parallel components that are not modelled by our cost model. The amount of parallelism is expected to increase which makes the prediction of the execution time of programs even more difficult in the future. Then the question how well a program is parallelizable becomes important.

References

[BFP+73] Manual Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[Car92] S. Carlsson. A note on HEAPSORT. *The Computer Journal*, 35:410–411, 1992.

[FJKT99] Ramzi Fadel, Kim V. Jakobsen, Jyrki Katajainen, and Jukka Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220:345–362, 1999.

[Flo64] Robert W. Floyd. Algorithm 245: TREESORT 3. *Communications of the ACM*, 7:701, 1964.

[FR75] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, Berlin/Heidelberg, 1998.

[HM91] Ryan Hayward and Colin McDiarmid. Average case analysis of heap building by repeated insertion. *Journal of Algorithms*, 12:126–153, 1991.

[Hoa61] C. A. R. Hoare. Algorithm 65: FIND. *Communications of the ACM*, 4:321–322, 1961.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, 2nd edition, 1996.

[HSR98] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms.* Computer Science Press, New York, 1998.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley Longman, Reading, 3rd edition, 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley Longman, Reading, 2nd edition, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, 2nd edition, 1988.

[KT97] Jyrki Katajainen and Jesper L. Träff. A meticulous analysis of mergesort programs. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, Berlin/Heidelberg, 1997.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, 1:Article 4, 1996.

[LL99] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[MR89] C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10:352–365, 1989.

[MS91] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP, Volume 1: Design & Efficiency*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1991.

[Ram94] Rajeev Raman. A simpler analysis of Algorithm 65 (Find). *SIGACT News*, 25:86–89, 1994.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6:233–247, 1993.

[San99] Peter Sanders. Fast priority queues for cached memory. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, volume 1619 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, Berlin/Heidelberg, 1999.

[Sed77] Robert Sedgewick. The analysis of Quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[Sed78] Robert Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21:847–857, 1978. Corrigendum, *ibidem 22* (1979), 368.

[Sib99] Jop F. Sibeyn. External selection. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 291–301. Springer-Verlag, Berlin/Heidelberg, 1999.

[Spo99] Maz Spork. Design and analysis of cache-conscious programs. Master's thesis, Department of Computing, University of Copenhagen, Copenhagen, 1999. Available at `http://www.diku.dk/research-groups/performance_engineering/`.

[SS95] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 44:1223–1235, 1995.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 3rd edition, 1997.

[Weg85] Lutz M. Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, C-34:362–367, 1985.

[Weg92] Ingo Wegener. The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$. *Information and Computation*, 97:86–96, 1992.

[Wil64] J. W. J. Williams. Algorithm 232: HEAPSORT. *Communications of the ACM*, 7:347–348, 1964.

Chapter 4

Set-associative caches and spatial locality

Abstract. The focus of this chapter is on the analysis of program behaviour on a *set-associative* cache, rather than on a fully associative cache as assumed in the preceding chapters. In general set-associative caches are difficult to analyse, and may exhibit rare and intense phenomena such as ping-pong effects, which can severely reduce performance. Some results from the literature are surveyed: a general analysis which explains why set-associative caches are difficult to deal with, and how much changes in the cache architecture can effect program performance, is presented, followed by two examples of analytical cache models which allows predictions to be made for certain kinds of programs. Programs which access memory mostly sequentially, and certain loop-nested array references, that occur frequently in numerical programs, are studied.

4.1 Introduction

In a direct-mapped cache each datum can be stored in only one location in the cache. Hence, any two datums associated with the same cache location, cannot exist in the cache at the same time. Because the mapping from memory locations to cache locations is done statically, without any knowledge of prior or future memory accesses, it is to be expected that a direct-mapped cache has a higher miss rate than a cache which uses, for example, the least recently used (LRU) replacement strategy. However, in an r-way set-associative cache each memory location can be stored in any of r different cache locations, allowing the cache a bit more freedom in the replacement strategy, but since r is typically small, this is still only a very crude approximation to the LRU strategy.

In the preceding chapters, it was assumed that a fully associative cache with the LRU replacement strategy was employed. Unfortunately most modern architectures implement set-associative caches with small sets. Typically, cache set sizes are in the range 1–8 lines even though exceptions do exist (see, for example, [Vet98] for the description of an 128-way set-associative cache design). Hence, one may question the validity of our results. On the other hand, our theoretical findings agrees remarkably well with measured performance. In this chapter some results from the literature are reviewed.

The structure of this chapter is as follows. The introduction is concluded with a formal definition of the behaviour of set-associative caches, followed by an analysis of one of the heap-construction programs from Chapter 3, which hints why the set-associativity part of the cache organisation is often abstracted away. Section 4.2 reviews some generally applicable results from [Tor98] which, unfortunately, are not strong enough to give any information on how a specific program performs. Further, in Section 4.3 and 4.4 two important classes of programs which have received special attention in the literature are studied. Section 4.3 investigates programs that access memory in several concurrent sequential memory traversals, while Section 4.4 analyses certain loop-nested array references, that occur frequently in numerical programs.

Set-associative caches

The principle behind a set-associative cache is to partition the cache lines into a number of (small) subsets. Each memory location is then associated with only one cache set thereby making the cache hardware simpler. The hope is that the memory accesses will be sufficiently distributed across the cache subsets to allow the many small cache sets to combine into a single large cache.

As in the preceding chapters we let M denote the size of the (combined) cache, measured in words, and B denote the size of the transfer unit between memory levels. Formally an **r-way set-associative cache** $\langle M, B, r\rangle$ behaves as follows:

- The M/B cache lines are addressed by a two part address, $\{(s, \ell) \mid s \in [0, \ldots, M/(Br)) \wedge \ell \in [0, \ldots r)\}$. The r cache lines selected by the s component of the address is referred to as a **cache set**.
- $C(b) = b \bmod (M/(Br)$ is a mapping of memory blocks onto the set $[0, \ldots, M/(Br))$.
- A memory block b can be placed only in cache lines $(C(b), j)$ where $j \in [0, \ldots r)$, i. e., in a single cache set. Replacement within a cache set can be done according to any online replacement strategy. Popular choices are LRU and FIFO replacement.

In the special case, when $r = 1$ and $r = M/B$, the cache is said to be **direct-mapped** and **fully associative**, respectively.

Heap-construction in a direct-mapped cache

In the reengineered repeated-insertion heap-construction method, we sort the elements on the path from each leaf node to the root. One path is handled at a time, so only $\lfloor \log_2 N \rfloor + 1$ nodes needs to be kept in the cache simultaneously to achieve good performance. Anyway, in a direct-mapped cache we cannot be certain that all elements on the path from a leaf node can coexist in the cache. We say that a node **conflicts** if it maps to the same cache line as any of its ancestors. Formally the node a conflicts if there exist an integer $i > 0$ such that $\lfloor a/B \rfloor \neq \lfloor a/(2^i B) \rfloor \wedge C(\lfloor a/B \rfloor) = C(\lfloor a/(2^i B) \rfloor)$ where C is defined as above. Our interest in conflicting nodes arises from the fact that in the worst case a conflicting node will be brought into the cache once for each of its descendants.

In the following theorem we show that in the worst case the repeated insertion heap-construction method will incur many more cache misses in a direct-mapped cache than it does in a fully associative cache.

Theorem 4.1 *Assume that the heap to be constructed is of size* $N = 2^h - 1 > M$ *for some integer* h*, i. e., the bottom level is full and the input does not fit into the cache. Furthermore, let the heap be perfectly aligned in memory so that the upper* $\log_2 B$ *levels fit in a single memory block, and assume that the cache is direct-mapped. Then will both the repeated-insertion method and the reenginered repeated-insertion method for constructing a heap incur at least*

$$(B \log_2 \frac{2M}{B})(\frac{N+1}{M} \log_2 \frac{N+1}{2M} + 1)$$

cache misses in the worst case.

Proof: Assume that a worst case input is given, i. e., an input where each sift-up operation proceeds all the way to the root. In a direct-mapped cache, a conflicting node of height h will be evicted from the cache before any reuse of that node can take place. Hence, it will miss once for each of its $2^{h+1} - 1$ descendants.

We start with the observation that none of the nodes on the upper $\log_2 M$ levels conflicts because the upper levels occupy only $M - 1$ consecutive locations in memory. A sift-up operation from some level of depth $\ell \geq \log_2 M$, will access one node on each level above it in the tree. In particular it will access a node on each of the upper $\log_2 M$ levels. The $\log_2 B$ upmost nodes resides in the same memory block while the the remaining $\log_2 M - \log_2 B$ nodes accessed resides in separate memory blocks, so a total of $\log_2 M - \log_2 B + 1 = \log_2(2M/B)$ blocks are accessed on the upper levels. As noted none of these blocks maps to the same cache line.

Now consider what happens during the scan over a level of depth $\ell \geq \log_2 M$. The level holds $kM = 2^{\ell - \log_2 M} M$ nodes. In each of the k segments there is M nodes, occupying M/B blocks, of which none conflicts with each other in the cache. Because of this, there must be at least one block (B nodes) which conflict with each of the $\log_2(2M/B)$ blocks accessed on the upper levels. Hence, there is a total of $B \log_2(2M/B) 2^{\ell - \log_2 M}$ conflicting nodes on level ℓ. Each conflicting node is the root of a subtree with $2^{\log_2(N+1) - \ell} - 1$ nodes, each of which causes the conflicting node to miss when sifted up. In total the number of cache misses incurred on the lower $\log_2(N+1) - \log_2 M$ levels is at least

$$\sum_{\ell = \log_2 M}^{\log_2 \frac{N+1}{2}} B \log_2 \frac{2M}{B} 2^{\ell - \log_2 M} (2^{\log_2(N+1) - \ell} - 1)$$

$$= (B \log_2 \frac{2M}{B})(\sum_{\ell = \log_2 M}^{\log_2 \frac{N+1}{2}} 2^{\log_2 \frac{N+1}{M}} - \sum_{\ell = \log_2 M}^{\log_2 \frac{N+1}{2}} 2^{\ell - \log_2 M})$$

$$= (B \log_2 \frac{2M}{B})(\frac{N+1}{M} \log_2 \frac{N+1}{M} - (\frac{N+1}{M} - 1))$$

$$= (B \log_2 \frac{2M}{B})(\frac{N+1}{M} \log_2 \frac{N+1}{2M} + 1)$$

□ This lower bound should be compared to the $N/B + o(N)$ upper bound which applies to the reengineered method when the cache is fully associative (see theorem 3.4. Hence, when the size of the cache is small relatively to the size of the heap, a direct-mapped cache will produce significantly more cache misses.

It is easy to reduce the worst-case cache-miss rate of the repeated-insertion methods in a direct-mapped cache by using extra memory: At each point in time, the elements on the path from the node currently under consideration to the root are kept in an auxiliary array. The array must be larger than $\lfloor \log_2 N \rfloor$ to be able to hold the elements, and it should be allocated in contiguous memory to guarantee that no conflicts exist within the array. We will later see that this strategy will reduce the number of cache misses to be close to the N/B lower bound for the reengineered variant, even in a direct-mapped cache. Note that this technique, collecting frequently used data in conflict free memory or registers, is a standard method for dealing with cache memories. The difficulty is in assessing the relative cost of the extra copying to the number of cache misses avoided.

The above analysis bring about several comments. First, the lower bound is not tight. In general the program can be expected to incur many more cache misses than the bound suggests, because conflicts within the lower levels of the tree are not considered. Unfortunately correcting this problem will complicate the calculations further. Second, calculating a similar bound in a cache with higher associativity seems complicated.

However incomplete the analysis is, it illustrates well the nature of the problems that must be managed when a set-associative cache is modelled. Obtaining information that is precise enough to decide if an optimisation (in this case copying) is worthwhile, often demands an unreasonable high level of detail in the analysis. Indeed, in the current form the analysis does not provide a clear answer.

4.2 Set-associative cache performance in general

In this section we present some general results on the performance of set-associative caches which suggest why cache performance is often characterised as being 'noisy' or 'inherently unstable'. The results are largely due to Torng [Tor98] and Denning [Den68].

4.2.1 Memory-access cost model

When analysing the goodness of paging strategies and access sequences we count both the number of memory references performed and the number of block transfers between the cache and memory, as recommended in [HP96, Tor98].

The total cost of an access sequence, $s = s_0, \dots, s_{n-1}$, in a memory hierarchy is calculated as $n\tau + m\tau^\star$ where m is the number of cache misses, τ is the cost of a memory reference, and $\tau^\star$ is the penalty for a cache miss. In practice we are mostly interested in comparing the relative cost of different access sequences, not in tuning the cache configuration, so only the ratio $\tau^\star/\tau$ is important. Hence, we normalise τ and $\tau^\star$ so that $\tau = 1$. In the context of caching a typical value for $\tau^\star$ is about $20 - 50$ for current hardware.

The motivation behind using this cost model rather than counting only the number of cache misses (page faults), as is done in much of the literature on paging and external memory algorithms (e. g., [MS91, ST85, RS89]), is that in the simpler model, unbounded speed-up, due, either to changes in the paging algorithm or to changes in the program behaviour, is possible. While in the weighted cost model no speed-up can be larger than $\tau^\star + 1$. Hence, the simpler model tend to overvalue memory tuning. Such one sided errors may be defendable in the context of paging where $\tau^\star$ is very large (10000 or more), but when the subject is cache memories, more precision is needed. This cost model is similar to the weighted pure C cost model that was employed in Chapter 3, except that the program has been abstracted away, so that only the memory access sequence is left.

We use the notation $A_r(s)$ to denote the cost of the memory-access sequence s in an r-way set-associative cache which employs the A replacement strategy within a cache set.

4.2.2 Working sets and working set sizes

Recall that the purpose of a cache is to take advantage of the reuse of data present in most programs. Data reuse may be classified into two categories: **temporal** reuse occurs when a datum is referenced more than once, while **spatial** reuse occurs when several data items residing in the same block in memory, are referenced. However, even if the amount of data reuse by a program is abundant, there is no guarantee that a cache will be able to reduce memory access time by much, since the cache replacement strategy may decide to evict most data items from the cache before the cache reuse can take place. To benefit from a cache, the program must also exhibit locality of reference; either **temporal** locality, in which the temporal reuse occurs within a reasonably short time frame; or **spatial** locality, in which the spatial reuse occurs within a short time frame.

To investigate the idea of locality, Denning [Den68] introduced the concept of the working set of a sequence of memory accesses, and the related concept of phases of execution, during which the working set remains stable. With these definitions it becomes possible to make quantitative statements about how much locality of reference is available in a memory trace.

In the definitions below we follow Torng's [Tor98] notation. The first three definitions, originated by Denning [Den68], explains how the decomposition of an access sequence into phases is handled, while the last definition, due to Torng [Tor98], introduce the average phase length as a measure of locality.

Definition 4.1 *For a sequence of memory references $s = s_0, \ldots, s_{n-1}$, let the **working set** $\mathrm{W}(s,i,j)$ be the set of distinct memory blocks in $s_i, \ldots, s_{j-1}$, i. e., $\mathrm{W}(s,i,j) = \{s_l | l \in \{i, \ldots, j)\}$.*

Definition 4.2 *For some constant k, let $I_{k,s}(i) = 0$ when $i = 0$ and let $I_{k,s}(i) = \max\{j | j \leq n \wedge |\mathrm{W}(s, I_{k,s}(i-1), j)| \leq k\}$ when $i > 0$.*

Definition 4.3 *Let j be such that $I_{k,s}(j) = n$, i. e., j is the largest value for which $I_{k,s}(j)$ is defined. A **k-decomposition** of s, $D_k(s)$, is defined to be the set of phases $P_i(s) = I_{i-1,k}(s), \ldots, I_{i,k}(s) - 1$ for $i \in \{1, \ldots, j\}$.*

Definition 4.4 *$L_k(s) = n/|D_k(s)|$, **the average phase length** $L_k(s)$ of $D_k(s)$. It should be noted that the simplified cost model ($\tau = 0$) is totally insensitive to changes in the average phase length, since it counts the number of cache misses only.*

We say that a sequence exhibit significant locality for working sets of size k if the average phase length is $L_k(s) \gg k$.

These definitions allow a derivation of the result (Lemma 3.1) by Sleator and Tarjan [ST85] which was applied extensively in the preceding chapters to prove the upper bounds on the number of cache misses incurred. Alternatively, one could prove, as Torng [Tor98] did, a similar result, which measures the competitiveness of the LRU-replacement strategy in the weighted cost model instead. Anyway, a perhaps more interesting result in [Tor98] is that if the adversery is restricted to choosing an access sequence which exhibits significant locality for working sets of size M, i. e., $L_M(s) = \Omega(\tau^\star M/B)$, then the LRU (and FIFO) replacement strategy will perform to within a constant factor of the optimal off-line strategy. Either way, it is clear that if the paging algorithm is LRU or FIFO, we have strong tools at our hands, to predict and tune the cache performance of programs. As we shall see the situation is not as favorable if a set-associative cache is employed.

4.2.3 General bounds on memory performance

In [Tor98] Torng analysed the effect on the memory access time, when increasing the associativity in an r-way set-associative cache.

The first result in [Tor98] is that an increase in set-associativity can greatly enhance the memory performance of a program.

Theorem 4.2 *Let $r \geq 1, t \geq 2$ be integers and let ϵ be an arbitrary real number such that $\epsilon > 0$. Then for any replacement strategy A which does not evict a block unless that block's set in the cache is full, there exists sequences s such that*

$$\frac{A_r(s)}{A_{tr}(s)} > \tau^\star + 1 - \epsilon$$

The essense of this theorem is that an increase in the set-associativity can improve memory performance of a program, from a miss on every memory reference, to a hit on every memory reference.

Another result from [Tor98] is perhaps a bit more surprising. It shows that memory performance may actually degrade when the set-associativity is increased from r to tr for an integer $t > 1$, if the system employs an online replacement strategy such as LRU or FIFO.

Theorem 4.3 *There exists an access sequence s such that*

$$\frac{\mathrm{LRU}_r(s)}{\mathrm{LRU}_{tr}(s)} < \frac{(t-1)r + \tau^\star + 1}{((t-1)r+1)(\tau^\star + 1)} + \epsilon$$

for all integers $r \geq 1, t > 1$ and all real numbers $\epsilon > 0$.

As an example of an access sequence for which a direct-mapped cache will perform better than a fully associative LRU cache, consider what happens when the memory locations $0, \ldots, M$ are scanned sequentially a number of times. The working set size of the access sequence is $M+1$ so it will not fit into the cache. In a direct-mapped cache two elements, e. g., 0 and M, will map to the same cache line, while the remaining elements will not be in any conflicts, so 2 cache misses will be incurred per $M+1$ memory references. On the other hand, it is clear that a fully associative LRU or FIFO cache will miss on every access on this sequence. However, if the cache employs the optimal longest-forward-distance off-line strategy within cache sets, it is obvious that $\mathrm{OPT}_r(s) \geq \mathrm{OPT}_{tr}(s)$ for any access sequence s since OPT_{tr} has the freedom to behave like OPT_r.

The main conclusion of these two results is that very little can be said about the performance of a set-associative cache in the general case. The only promise that is given by an r-way cache is that it will perform no worse than a cache of size r, which is essentially no guarantee at all. The potential for much better results exists, but to decide if a specific program runs well in a set-associative cache, must be determined through a detailed analysis of the mapping of memory accesses into the cache. Experimental results such as the $2:1$ rule [HP96], which states that a direct-mapped cache of size M has about the same miss rate as a 2-way set-associative cache of size $M/2$ suggest that most programs behave reasonably well in a set-associative cache, but this is of course of little interest when dealing with a specific program.

4.3 Concurrent sequential memory traversal

A widely used technique in the design of external memory algorithms is to express the algorithm in terms of a number of concurrent sequential memory traversals. The idea is to take advantage of spatial locality. This class of algorithms can be generalised into the following schema: k arrays of total length N are to be scanned concurrently and sequentially. When an array has been accessed, the pointer into that array is advanced one position, so that each element is accessed exactly once. We label the number of elements scanned in the sequence j by N_j. Hence, $N = \sum_{j=1}^{k} N_j$. For the purpose of the analysis an adversery is introduced which decides in each iteration the array which should be accessed next.

In the general form of the problem, studied in [San99], the adversery is allowed to schedule the sequences without restrictions, including the freedom to specify the length of each sequence N_j. More precisely, the only limitations the adversery must obey is that the total length of the traversals and the number of concurrent traversals must be N and k respectively. It should also be noted that the requirement, that each element is accessed only once, is essential for the results obtained.

In a hierarchical memory it is a simple matter to specify a cache replacement strategy that allows concurrent sequential traversals to execute with a miss rate equal to the lower bound N/B if only the number of sequences, k, is less than the capacity of the cache measured in blocks, $m = M/B$. If on the other hand the cache is implemented as a set-associative cache, the adversery has no difficulty to force a cache miss on every access if only the number of concurrent traversals is larger than the associativity of the cache, r.

4.3.1 Results

The technique studied in [San99] to improve on this situation is to deny the adversery access to information about how the sequences are mapped into the cache by randomizing the starting address x_j of each sequence so that the values $x_j \bmod (m/r)$ are uniformly distributed independent random variables. To simplify the discussion only direct-mapped caches are considered in the following, but it should be noted that similar results for caches with a higher associativity are available in [San99].

Theorem 4.4 *Given a direct-mapped cache with m cache lines each of size B elements, and k arrays of total length N which are to be scanned sequentially. Let $U(X)$ denote the expected number of cache misses for the schedule X if the starting addresses of the k arrays accessed by X are randomized. Further, let $U = \max_X U(X)$ for all possible schedules of the k sequences. Given these definitions U is bounded from above:*

$$U \leq k + \frac{N}{B} + (B-1)\frac{k}{m}\frac{N}{B}$$

This bound is supplemented with a similar lower bound which shows that the upper bound is in some sense tight.

$$U \geq \frac{N}{B} + (B-1)\frac{k-1}{m+k-1}\frac{N}{B}$$

In the upper bound the N/B term represents the number of compulsory misses, while the $(B-1)(k/m)(N/B)$ term represents the extra cache misses incurred due to interference.

One interesting application of this result is that it allows prediction of how many concurrent sequential traversals that a certain cache configuration will support with a constant bounded penalty. If, for example, we are willing to accept a penalty of no more than $2N/B$ cache misses, i. e., twice the lower bound, we see that the number of concurrent traversals must be less than $m/(B-1)$.

To make the result in Theorem 4.4 practical, it was further generalized to allow random access to a moderately sized work space concurrently with the sequence traversal. The work space can be used to hold additional data, such as sequence pointers and intermediate results.

Theorem 4.5 *Let U be defined as above in Theorem 4.4 and let w be the size, measured in cache lines, of the work space allocated in conflict free memory. Then the expected number of cache misses incurred during N accesses to sequence data, and any number of accesses to the work space is bounded by*

$$w + (1 - \frac{w}{m})U + 2\frac{w}{m}N$$

Hence, if the work space is substantially smaller than the cache, very few extra cache misses will be expected due to accesses to the work space.

With this result it is possible to obtain an estimate on the number of cache misses incurred by the repeated-insertion method, if the elements on the path to the root are kept in an auxiliary array as was suggested in introduction to this chapter: If we assume an 8 Kbyte direct-mapped cache with a cache line size of 32 bytes, an element size of 4 bytes, and a heap of size $N < 2^k - 1$ for $k = 24$; ($B = 8$, $m = 256$). Then we get from Theorem 4.4 that $U < 1.67N/B$ and further we get from Theorem 4.5 that the expected number of cache misses is less than $1.84N/B$. Note, however, that this result requires that the starting address of each level in the tree is randomized. We claim, without a proof, that in this particular case the randomization is not necessary since the access pattern is fixed so that no two memory traversals are allowed to proceed at the same rate through memory.

4.4 Predicting memory performance for loop-nested array references

Much of the literature on analytical cache models has its background in numerical programs from the scientific computing community. Many numerical programs consist of nested loop structures, containing array references dependent on the loop iteration variables only. Due to the regular memory-access pattern generated by this kind of loops, the cache behaviour may be modelled analytically in many cases. The goal of the model is to provide a precise estimate on the cache behaviour of a program rapidly enough so that the estimate can be utilised during program compilation, or even at run-time, to direct optimisation. A study in this direction is [TFJ94] by Temam et al. which provides the model for the following discussion.

4.4.1 Problem description

In this section we investigate the cache performance of affine loop-nested array references in a direct-mapped cache. More precisely, we are interested in loop-nestings normalised so that the loop variable counts upward from zero in steps of one, for a constant number of iterations. That is, the iteration count may only depend on the *size* of the problem, not on the input values. The loops in a nesting are numbered $1, \ldots, n$; loop 1 being the outermost loop and loop n being the innermost loop. The iteration count of loop level i is labelled $\mathcal{N}_i$ and the corresponding loop variable is labelled j_i. Furthermore, all array references must be of the form

$$\mathbf{A}[\alpha_1 j_{\gamma_1} + \beta_1, \ldots, \alpha_m j_{\gamma_m} + \beta_m]$$

where $\mathbf{A}$ is the name of the array, m is the number of dimensions of the array and $\alpha_k, \beta_k, \gamma_k$ are constants. When written in pointer notation, such a reference take on the form of a linear expression: $\mathbf{A}_0 + \mathbf{A}_1 j_1 + \ldots + \mathbf{A}_n j_n$ where $\mathbf{A}_0, \ldots, \mathbf{A}_n$ are constants. An example of such a loop is seen in Figure 4.1.

In order to simplify the discussion we will make some rather unrealistic assumptions: We analyse the effect of each array reference[1] in isolation. That is, we only consider the possibility of reuse of array elements that was earlier accessed by the same array reference, and likewise we do not consider the possibility of interference between different array accesses. Furthermore we will

[1] In most of this thesis a 'reference' is a run time entity. In this section however the source program is analysed statically, so here an array reference exist in the domain of the program source.

initially assume that the size of a cache line is equal to the word size. Especially this last assumption is imprecise. While extending the model to include these effects is by no means trivial, it can in some sense be considered to be 'more of the same'.

The strategy for evaluating the miss count of a loop nesting is to count, for each array reference in the source program, the number of cache misses due to interference. This is done in two separate steps: The first step seeks to identify the data reused by the program, and the time ordering of that reuse. The memory locations of the reused data must be determined in sufficient detail to map it into the cache. Mapping the data into the cache is taken care of by the second step, which exploits the rigid structure of the loop nesting and the resulting reuse set, to avoid most considerations for the time ordering of references.

4.4.2 Self reuse and self interference

The first step in the analysis is to determine, for each array reference, the loop level at which reuse occurs the first time — **the reuse loop level** — and further to determine the array elements which are referenced during a single iteration on that level.

for $j_1 := 0$ **to** $\mathcal{N} - 1$
  for $j_2 := 0$ **to** $\mathcal{N} - 1$
    register $:= \mathbf{X}[j_1][j_2]$
    for $j_3 := 0$ **to** $\mathcal{N} - 1$
      $\mathbf{Z}[j_1][j_3] := \mathbf{Z}[j_1][j_3] + \mathit{register} \times \mathbf{Y}[j_2][j_3]$

Figure 4.1: Sub-matrix multiplication

As an example consider the loop nesting in Figure 4.1 which multiplies two $\mathcal{N} \times \mathcal{N}$ sub-matrices of some larger matrices. The reuse of the reference $\mathbf{Z}[j_1][j_3]$ occurs in loop 2 while the reuse of $\mathbf{Y}[j_2][j_3]$ occurs in loop 1. The reuse of the reference $\mathbf{X}[j_1][j_2]$ is exploited explicitly by loading it into a register, so it will not concern us further.

Having determined the reuse loop level ℓ of a reference R, **the theoretical reuse set**, $\mathrm{TRS}_\ell(R)$ is defined to be the set of array elements that are to be reused by R; i. e., the theoretical reuse set consist of the array elements accessed in a single iteration of the reuse loop. For example, the theoretical reuse set for the reference $\mathbf{Y}[j_2][j_3]$ is equal to $\{\mathbf{Y}[0][0], \ldots, \mathbf{Y}[\mathcal{N}-1][\mathcal{N}-1]\}$ while the reuse set for the reference $\mathbf{Z}[j_1][j_3]$ is equal to $\{\mathbf{Z}[j_1][0], \ldots, \mathbf{Z}[j_1][\mathcal{N}-1]\}$.

4.4.3 Evaluating the cache footprint of the theoretical reuse set

The second step in the analysis is to calculate the effect of the TRS in the cache. To this end the **actual reuse set** is introduced. The actual reuse set of reference R, contains exactly those locations in the cache which are referenced only once when the TRS is mapped into the cache. With this information at hand, the number of cache misses incurred during a single iteration of the reuse loop for reference R, can be calculated as

$$|\mathrm{TRS}_\ell(R)| - |\mathrm{ARS}_\ell(R)|.$$

The crucial observation that allows a TRS to be mapped into the cache without reverting to simulation is that the structure of a TRS is very regular. A TRS is composed of a number, N, of segments, each of size S, which are spaced across the array with a fixed distance δ in between segments. Hence, the shape of a TRS can be characterised by the three parameters $\langle N, S, \delta\rangle$;

This structure can be mapped into the cache by a procedure which is essentially equivalent to the calculation of $\gcd(\delta, M)$ by Euclids algorithm. Consider any reuse set $\mathrm{TRS}_\ell(R) = \langle N, S, \delta\rangle$. We can without loss of generality assume that the first segment in the TRS maps to the cache locations $\{0, \ldots, S \bmod M)$; then the next segment must map to locations $\{\delta \mod M, \ldots, (\delta + S) \bmod M)$, and generally the ith segment maps to locations $\{i\delta \mod M, \ldots, (i\delta + S) \bmod M)$. This procedure is equivalent to the steps taken by the blockwise dolphin method for the segment interchanging problem, which was analysed in detail in Section 2.4.2, and so the precise position of interference and consequently also the amount of cache interference can be predicted in a similar way as the blockwise dolphin method uses to avoid processing a position twice. Hence, the cost when mapping the TRS to the cache is proportional to calculating $\gcd(\delta, M)$ which has a running time of $O(\log M)$. We see that the cache miss rate can be estimated much faster and in much less space than a simulation would require.

4.4.4 Expanding the model

The most questionable assumption made in the above was the assumption that the cache line size is equal the the word size, i. e., $B = 1$. If $B > 1$ it becomes necessary to measure the size of the TRS and the ARS in cache lines rather than words, and further the regular shape of the TRS used to map it into the cache will be an approximation only. Neither modification represents a serious problem however.

In the PhD-thesis of John Harper [Har99] and in [HKN99] this basic method was further elaborated to take other sources of reuse and interference into account through an extensive hierarchy of cache reuse and interference types which include both temporal and spatial effects. Moreover, the cache model was expanded to handle caches with associativity larger than 1 as well.

4.5 Concluding remarks

While prediction of the behaviour of a program in a set-associative cache is often hard, it is quite simple to modify programs to incur few cache misses if the underlying algorithm exhibits good locality for a working set size substantially smaller than the cache. By copying data to an auxiliary array, a set-associative cache can always be made behave almost like a cache with replacement under program control. The challenge is to assess if the cache behaviour of the original program is bad enouh to justify the extra cost due to copying. Several fairly accurate analytical cache models exist in restricted program domains, but the results of Torng [Tor98] suggest that it is unreasonable to expect to come up with a single unified cache model that will be able to handle all programs.

Indeed, none of the currently available analytical methods are able to provide a prediction for the seemingly simple program to construct a heap, that was used as an example in this chapter. However, some optimisations such as padding array dimensions, and randomised data placement are essentially free of runtime overhead, and does not require an accurate prediction to be applied.

Assuming that a unified cache model is not viable, the classification of programs becomes important. One approach for program classification is based on the observation that while the performance of set-associative caches is inherently unstable, some programs are much more sensitive to minor changes in data layout than others. It seems that no specific studies in this direction exist, but the idea is implicit in the concept of a 'translation group' which is included as a component in the interference hierarchy exploited in [Har99]. The attraction of analysing the stable programs only is that lower/upper bound results can be more accurate.

References

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11:323–333, 1968.

[Har99] John S. Harper. *Analytic Cache Modelling of Numerical Programs*. PhD thesis, University of Warwick, Coventry, UK, 1999. Available at `http://www.dcs.warwick.ac.uk/people/research/John.Harper/`.

[HKN99] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, 1999. Available at `http://www.dcs.warwick.ac.uk/people/research/John.Harper/`.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, 2nd edition, 1996.

[MS91] L. L. McGeoch and Daniel D. Sleator. Memory versus randomization in on-line algorithms. *Algorithmica*, 6:816–825, 1991.

[RS89] Prabhakar Raghavan and M. Snir. Memory versus randomization in on-line algorithms. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Compute Science*, pages 687–703, Berlin/Heidelberg, 1989. Springer-Verlag.

[San99] Peter Sanders. Accessing multiple sequences through set associative caches. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 655–664. Springer-Verlag, Berlin/Heidelberg, 1999. An extended version is available at `http://www.mpi-sb.mpg.de/~sanders/`.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS*, pages 261–271, 1994.

[Tor98] E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175–200, 1998.

[Vet98] Scott Vetter, editor. *POWER3 Introduction and Tuning Guide*. Redbooks. IBM Corporation, Technical Support Organization, 1998. Available at `http://www.ibm.redbooks.com/`.

Chapter 5

Final conclusions

During the course of this work, two simple problems was studied in detail. Swapping two adjacent segments of an array, and that of constructing an implicit binary heap. In both cases existing implementations were improved by applying classic program transformation techniques to reschedule memory operations for better locality, and also, in one case, sampling was used to reduce both the instruction count and the number of memory references executed. Further it also became necessary to apply formal program development and verification techniques to ensure correctness of the developed programs.

The main steps in the tuning process were; – first to tune for locality; – next to tune for instruction count. In that order. Often did the improvements in running time not manifest themselves until at the end of the process, but if the programs were tuned with just instruction count in mind, only very little could be gained. Hence the memory tuning is an essential part of the process. A corollary to this observation is that invoking the highest levels of compiler optimizations are of little benefit if the program does not exhibit significant locality.

One conclusion that can be drawn from this work, is that the cost model used by the programmer can have a profound effect on the resulting programs. A common perception is that the memory hierarchy is a machine dependent architectural detail that is best abstracted away, but this can result in programs that perform very poorly, as was observed for the dolphin method. However, the opposite approach, applied in the design of many external memory algorithms, that only external memory events are counted, is equally flawed. Hence, both instructions executed and cache misses must be counted.

Another relevant lesson learned during the case studies, was that cache optimizations can mostly be done in a machine independent way. Caches are universally present in todays machines, and have a sufficiently similar structure so that details can be safely ignored. Further, temporal locality is a property that can be measured independently of cache structure and data representation, while measuring spatial locality requires knowledge of the data representation, but still does not need information about cache structure. It is only when quantitative predictions about the number of cache misses or memory access times are made, the structure of the cache must be known.

The exception to this rule was studied in Chapter 4. Set-associative caches, and in particular direct-mapped caches, are difficult to deal with since, even if a program exhibit significant locality, the cache may not be able to take advantage of this. In particular when memory is accessed in a strided pattern, the performance depends strongly on the associtivity of the cache, the size of the stride and on the exact alignment of data structures in memory. However, such access patterns can often be avoided by copying data to a work area.

Directions for further work

In brief, locality means that subsequent accesses to a data element occurs close in time. However in the parallel processing community such a program behaviour is called a data dependency, which can prevent parallel execution. Hence, there is a tension between parallelising a program and coding for locality. It was the intent that part of this thesis would have investigated this area, but it is now left for further studies. For the time being, though, it seems that if only instruction-level parallelism is considered, the effect is not significant.

On a related subject, our cost model describes the memory bottleneck entirely as a latency issue. On high-end machines, however, much parallelism is available in the memory subsystem as well, and for many programs, performance is hindered not so much by latency to memory but by the lack of bandwidth.

Bibliography

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[Ben86] Jon Bentley. *Programming Pearls.* Addison-Wesley, Reading, 1986.

[BFP+73] Manual Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[Car92] S. Carlsson. A note on HEAPSORT. *The Computer Journal*, 35:410–411, 1992.

[DD81] K. Dudziński and A. Dydek. On stable minimum storage merging algorithm. *Information Processing Letters*, 12:5–8, 1981.

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11:323–333, 1968.

[Den80] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6:64–84, 1980.

[Du88] S. Dvořák and B. Ďurian. Merging by decomposition revisited. *The Computer Journal*, 31:553–556, 1988.

[FJKT99] Ramzi Fadel, Kim V. Jakobsen, Jyrki Katajainen, and Jukka Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220:345–362, 1999.

[Flo64] Robert W. Floyd. Algorithm 245: TREESORT 3. *Communications of the ACM*, 7:701, 1964.

[FR75] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[FS66] W. Fletcher and R. Silver. Interchange of two blocks of data. *Communications of the ACM*, 9:326, 1966.

[GKP99] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 1999.

[GM81] David Gries and Harlan Mills. Swapping sections. Technical report, Department of Computer Science, Cornell University, Ithaca NY 14853, 1981.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, Berlin/Heidelberg, 1998.

[Har99] John S. Harper. *Analytic Cache Modelling of Numerical Programs*. PhD thesis, University of Warwick, Coventry, UK, 1999. Available at `http://www.dcs.warwick.ac.uk/people/research/John.Harper/`.

[Hil87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987.

[HKN97] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Predicting the cache miss ratio of loop-nested array references. Technical report, University of Warwick, Coventry, UK, December 1997.

[HKN99] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, 1999. Available at `http://www.dcs.warwick.ac.uk/people/research/John.Harper/`.

[HM91] Ryan Hayward and Colin McDiarmid. Average case analysis of heap building by repeated insertion. *Journal of Algorithms*, 12:126–153, 1991.

[Hoa61] C. A. R. Hoare. Algorithm 65: FIND. *Communications of the ACM*, 4:321–322, 1961.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, 2nd edition, 1996.

[HSR98] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, New York, 1998.

[Knu93] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, Reading, 1993.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman, Reading, 3rd edition, 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman, Reading, 2nd edition, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, 2nd edition, 1988.

[KT97] Jyrki Katajainen and Jesper L. Träff. A meticulous analysis of mergesort programs. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, Berlin/Heidelberg, 1997.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, 1:Article 4, 1996.

[LL99] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[MR89] C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10:352–365, 1989.

[MS87] John L. Mohammed and Carlos S. Subi. An improved block-interchange algorithm. *Journal of Algorithms*, 8:113–121, 1987.

[MS91a] L. L. McGeoch and Daniel D. Sleator. Memory versus randomization in on-line algorithms. *Algorithmica*, 6:816–825, 1991.

[MS91b] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP, Volume 1: Design & Efficiency.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1991.

[Ram94] Rajeev Raman. A simpler analysis of Algorithm 65 (Find). *SIGACT News*, 25:86–89, 1994.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6:233–247, 1993.

[RS89] Prabhakar Raghavan and M. Snir. Memory versus randomization in on-line algorithms. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Compute Science*, pages 687–703, Berlin/Heidelberg, 1989. Springer-Verlag.

[San99a] Peter Sanders. Accessing multiple sequences through set associative caches. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 655–664. Springer-Verlag, Berlin/Heidelberg, 1999. An extended version is available at `http://www.mpi-sb.mpg.de/~sanders/`.

[San99b] Peter Sanders. Fast priority queues for cached memory. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, volume 1619 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, Berlin/Heidelberg, 1999.

[Sed77] Robert Sedgewick. The analysis of Quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[Sed78] Robert Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21:847–857, 1978. Corrigendum, *ibidem 22* (1979), 368.

[She97] Ching-Kuang Shene. An analysis of two in-place array rotation algorithms. *The Computer Journal*, 40(9):541–546, 1997.

[Sib99] Jop F. Sibeyn. External selection. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 291–301. Springer-Verlag, Berlin/Heidelberg, 1999.

[Spo99] Maz Spork. Design and analysis of cache-conscious programs. Master's thesis, Department of Computing, University of Copenhagen, Copenhagen, 1999. Available at `http://www.diku.dk/research-groups/performance_engineering/`.

[SS95] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 44:1223–1235, 1995.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 3rd edition, 1997.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS*, pages 261–271, 1994.

[Tor98] E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175–200, 1998.

[Vet98] Scott Vetter, editor. *POWER3 Introduction and Tuning Guide*. Redbooks. IBM Corporation, Technical Support Organization, 1998. Available at `http://www.ibm.redbooks.com/`.

[Weg85] Lutz M. Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, C-34:362–367, 1985.

[Weg92] Ingo Wegener. The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$. *Information and Computation*, 97:86–96, 1992.

[Wil64] J. W. J. Williams. Algorithm 232: HEAPSORT. *Communications of the ACM*, 7:347–348, 1964.

Appendix A

Program listings

A.1 Segment interchanging

stl/dolphin/rotate.cc

```
#include <algorithm>  // defines std::__rotate

namespace STL_Dolphin {
  using std::__rotate;

  template <class Rnd>
    inline void rotate (Rnd first, Rnd middle,
                        Rnd last) {
      typedef  typename iterator_traits<Rnd>::difference_type Distance;
      if (first == middle || middle == last) return;
      __rotate (first, middle, last, (Distance*)0, random_access_iterator_tag());
    }
}
```

tuned/reverse/rotate.cc

```
/*
 *
 * Segment Interchange.
 * Interchange array segments A and B:  AB -> BA
 *
 * Interchange is done by exploiting
 * reverse(reverse(A) reverse(B)) = BA
 * Optimised by reordering the swap operations.
 *
 * References: Part of the folklore.
 *
 */

#include <iterator>
#include <algorithm> //Defines iter_swap

namespace Tuned_Reverse {
  using std::iterator_traits;
  using std::iter_swap;

  template <class Rnd>
    void inline rotate4(Rnd a, Rnd b, Rnd c, Rnd d) {
      iterator_traits<Rnd>::value_type tmp = *a;
      *a = *b; *b = *c; *c = *d; *d = tmp;
    }

  template <class Rnd>
    void inline rotate3(Rnd a, Rnd b, Rnd c) {
      iterator_traits<Rnd>::value_type tmp = *a;
      *a = *b; *b = *c; *c = tmp;
    }
```

```
  template <class Rnd>
    void rotate(Rnd first, Rnd middle, Rnd beyond) {
      Rnd i, j, k, l;

      for (i = first, j = middle - 1, k = middle, l = beyond - 1;
           i < j && k < l;
           i++, j--, k++, l--)
        rotate4(i, k, l, j);

      for ( ;k < l; i++, k++, l--)  // At most one of these two loops will
        rotate3(i, k, l);           // be executed.
      for ( ;i < j; i++, j--, l--)
        rotate3(i, l, j);

      for ( ;i < l; i++, l-- )
        iter_swap(i, l);
    };
}
```

tuned/dolphin/rotate.cc

```
/*
 *
 * Segment Interchange.
 * Interchange array segments A and B:  AB -> BA
 *
 * Interchange is done by exploiting the additive subgroup.
 * Optimised by processing the array blockwise
 *
 * References: Chapter 2.
 *
 */

#include <assert.h>
#include <iterator>
#include <algorithm>  //defines std::copy

namespace Tuned_Dolphin {
  using std::copy;

  template <class Rnd>
    class segment_buffer {
    public:
      typedef std::iterator_traits<Rnd>::value_type element;
      typedef std::iterator_traits<Rnd>::difference_type distance;
      const distance MaxBufSize = CACHE_SIZE / (4 * sizeof(element));

    protected:
      Rnd first;
      distance size;
      element buf[MaxBufSize];

    public:
      void buffer() {};

      inline void load(Rnd src_first, distance src_size) {
        assert((src_size <= MaxBufSize) && (src_size >= 0));
        first = src_first; size = src_size;
        copy(first, first + size, buf);
      }

      inline void unload(Rnd src_first, Rnd out_first, distance out_size) {
        if (out_size == 0) return;
        assert(out_size <= size);
        if (src_first == first) {
          copy(buf, buf + out_size, out_first);
        } else {
          assert(src_first + out_size == first + size);
          copy(buf + (src_first - first),
               buf + size,
               out_first);
        }
      }
```

```
    };

#define T(x) (((x) + m) % N)

  template <class Rnd, class distance>
    inline distance rotate_hole(Rnd a, int m, int N, distance hole,
                         int hole_size, int steps) {
    for (;steps > 0; steps--, hole = T(hole))
      copy(a + T(hole), a + T(hole) + hole_size, a + hole);
    return hole;
  };

  template <class Rnd>
    void rotate(Rnd first,
               Rnd middle,
               Rnd beyond) {
    typedef std::iterator_traits<Rnd>::difference_type distance;

    segment_buffer<Rnd> buffer;
    segment_buffer<Rnd> extra_buffer;

    distance hole;
    const distance N = beyond - first;
    const distance m = middle - first;
    int i, left, right, leftcost, rightcost;
    int leftbufsz, rightbufsz, smallbufsz;

    /* Follow the gcd calculation to determine block sizes
       and number of blocks to move between buffer unloads/loads */
    left = m; right = N - m;
    leftcost = rightcost = 1;
    while (left > buffer.MaxBufSize && right > buffer.MaxBufSize) {
      if (left > right) {
        rightcost = (left / right) * leftcost + rightcost;
        left = left % right;
      } else {
        leftcost = (right / left) * rightcost + leftcost;
        right = right % left;
      }
    };

    if (left > right) {
      if (right != 0) {
        rightbufsz = (buffer.MaxBufSize / right) * right;
        leftbufsz = left % right;
      } else {
        rightbufsz = buffer.MaxBufSize;
        leftbufsz = 0;
      };
      smallbufsz = left % rightbufsz;

      hole = N - right;
      buffer.load(first + hole, right);
      for (i = left / rightbufsz; i > 0; i--) {
        hole -= rightbufsz - right;
        assert(hole == N - ((left / rightbufsz) - i + 1) * rightbufsz);
        extra_buffer.load(first + hole, rightbufsz - right);
        hole = rotate_hole(first, m, N, hole, rightbufsz, leftcost - 1);
        extra_buffer.unload(first + T(hole + right),
                            first + hole + right,
                            rightbufsz - right);
        hole = rotate_hole(first, m, N, hole, right, 1);
      };
      assert(hole == N - (left / rightbufsz) * rightbufsz - right);
      hole -= smallbufsz - leftbufsz;
      extra_buffer.load(first + hole, smallbufsz - leftbufsz);
      hole = rotate_hole(first, m, N, hole,
                         smallbufsz - leftbufsz + right, rightcost - 1);
      assert(hole == N - m - right + leftbufsz);
      buffer.unload(first + T(hole), first + hole, right - leftbufsz);
      hole += right - leftbufsz;
      hole = rotate_hole(first, m, N, hole, smallbufsz, leftcost - rightcost);
      assert(T(hole) == N - right - left);
      extra_buffer.unload(first + T(hole + leftbufsz),
```

```
                        first + hole + leftbufsz,
                        smallbufsz - leftbufsz);
      if (leftbufsz != 0) {
        hole = rotate_hole(first, m, N, hole, leftbufsz, rightcost);
        assert(T(hole) == N - right);
        buffer.unload(first + T(hole), first + hole, leftbufsz);
      }
    } else { /* right >= left */
      if (left != 0) {
        leftbufsz = (buffer.MaxBufSize / left) * left;
        rightbufsz = right % left;
      } else {
        leftbufsz = buffer.MaxBufSize;
        rightbufsz = 0;
      };
      smallbufsz = right % leftbufsz;

      hole = 0;
      buffer.load(first + hole, left);
      for (i = right / leftbufsz; i > 0; i--) {
        assert(hole == ((right / leftbufsz) - i) * leftbufsz);
        extra_buffer.load(first + hole + left, leftbufsz - left);
        hole = rotate_hole(first, m, N, hole, leftbufsz, rightcost - 1);
        assert(T(hole) == ((right / leftbufsz) - i) * leftbufsz + left);
        extra_buffer.unload(first + T(hole), first + hole, leftbufsz - left);
        hole += leftbufsz - left;
        hole = rotate_hole(first, m, N, hole, left, 1);
      };
      assert(hole == (right / leftbufsz) * leftbufsz);
      extra_buffer.load(first + hole + left, smallbufsz - rightbufsz);
      hole = rotate_hole(first, m, N, hole,
                         smallbufsz - rightbufsz + left, leftcost - 1);
      assert(T(hole) == (N - smallbufsz) % N);
      buffer.unload(first + T(hole + smallbufsz),
                    first + hole + smallbufsz,
                    left - rightbufsz);
      hole = rotate_hole(first, m, N, hole, smallbufsz, rightcost - leftcost);
      assert(T(hole) == (right - smallbufsz + left) % N);
      extra_buffer.unload(first + T(hole), first + hole, smallbufsz - rightbufsz);
      hole += smallbufsz - rightbufsz;
      if (rightbufsz != 0) {
        assert(T(hole) == right + left - rightbufsz);
        hole = rotate_hole(first, m, N, hole, rightbufsz, leftcost);
        assert(T(hole) == left - rightbufsz);
        buffer.unload(first + T(hole), first + hole, rightbufsz);
      }
    }
  }
}
```

driver.cc

```
#include <iostream>   // defines cout and cerr
#include <stdlib.h>
#include <time.h>     // defines clock_t and clock
#include "rotate.cc"  // defines NAME::rotate

typedef int element;

#ifdef NDEBUG
#define IT 10
#define MAXSIZE ((1 << 25) / sizeof(element))
#else
#define IT 1
#define MAXSIZE 800
#endif

element ab[MAXSIZE];

using namespace std;

int main(int argc, char **argv) {
  int size, mid;
  clock_t c_start, c_stop, e_time;
```

```
  for (int i = 0;i < MAXSIZE; i++) ab[i] = (element) i;

  size = atoi(argv[1]);
  mid = atoi(argv[2]);

  if (argc != 3) {
    cerr << "Usage: " << argv[0] <<" <size> <midpoint>\n";
    exit(1);
  }

  if (size > MAXSIZE || size <= 0) {
    cerr << "Size out of range: 0<" << size << "<" << MAXSIZE << endl;
    exit(1);
  };

  if ( mid > size || mid <= 0 ) {
    cerr << "midpoint out of range: 0<" << mid << "<" << size << endl;
    exit(1);
  };

  // The timing loop
  c_start = clock();
  for (int i = 0; i < IT; i++)
    for (element *a = ab; a + size <= ab + MAXSIZE; a += size)
      NAME::rotate(a, a + mid, a + size);
  c_stop = clock();
  e_time = c_stop - c_start;

  /* Check the results */
  for (element *a = ab; a + size <= ab + MAXSIZE; a += size) {
    for (int i = 0; i < size; i++) {
      element expected = (a - ab) + (i + IT * mid) % size;
      if (a[i] != expected) {
        cerr << "Fault: a["
             << i
             << "] = "
             << a[i]
             << "; expected "
             << expected
             << endl;
        break;
      }
    };
  }

  cout << mid
       << "\t"
       <<  (1e9 * e_time) / ((double)(MAXSIZE / size * size) *
                             (double)IT *
                             (double)CLOCKS_PER_SEC)
       << endl;

  return(0);
}
```