Performance engineering case study: heap construction

Jesper Bojesen, Jyrki Katajainen, and Maz Spork

Department of Computing, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark; Tel: (+45) 35 32 14 00; Fax: (+45) 35 32 14 01

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.

Categories and Subject Descriptors: E.1 [**Data Structures**]; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Sorting and searching

General Terms: Algorithms, Experimentation, Performance, Theory

Additional Key Words and Phrases: Binary heaps, memory tuning, code tuning

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

A preliminary version of this work was presented at the 3rd International Workshop on Algorithm Engineering, London, July 1999.
Part of this work was done while the second author was visiting the Department of Computer Science at the University of Turku, Finland.
Supported in part by the Danish Natural Science Research Council under contracts 9701414 (project Experimental Algorithmics) and 9801749 (project Performance Engineering).
Name: Jesper Bojesen
Affiliation: UNI-C, Danish Computing Centre for Research and Education
Address: Technical University of Denmark, Building 304, DK-2800 Lyngby, Denmark
Name: Jyrki Katajainen
Affiliation: Department of Computing, University of Copenhagen
Address: Universitetsparken 1, DK-2100 Copenhagen East, Denmark
Name: Maz Spork
Affiliation: VISIONIK a/s
Address: Struenseegade 9, 3. sal, DK-2200 Copenhagen North, Denmark
Technical Report 99/12, Department of Computing, University of Copenhagen, Copenhagen, Denmark (1999).

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., [Carlsson 1992]).

Another tradition popularized by Knuth (see, e. g., [Knuth 1998]) 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 [Williams 1964] 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., [LaMarca and Ladner 1996; Sanders 1999]) and heapsort based on a binary heap is not the fastest sorting method (see, e. g., [Moret and Shapiro 1991; LaMarca and Ladner 1999]), 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 [1964] and Floyd [1964]. 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 [Hayward and McDiarmid 1991], 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 LaMarca and Ladner [1996].

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 [1999]. A fast external-memory algorithm for heap construction have been presented by Fadel et al. [1999]. 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 2. The results of a micro benchmark that give some motivation for the cost model are reported in Section 2.4, but it can be skipped in the first reading. The notation used is established in Section 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 3.2 which can be skipped in the first reading. The known heap-construction methods are described and analysed in Section 4 and our reengineered versions in Section 5. The experimental results on the performance of the heap-construction programs are reported in Section 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 7.

2. METICULOUS ANALYSIS

In meticulous analysis, the term coined by Katajainen and Träff [1997], 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., [Knuth 1998]). In this section we define the models of computation and cost, under which we carry out a meticulous analysis of heap-construction programs.

2.1 Primitive operations

We use the standard **word random-access machine** (RAM) (see, e. g., [Hagerup 1998]) 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++ [Stroustrup 1997] and compiled to pure C [Katajainen and Träff 1997]. 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` $\lhd$ `z) goto` λ", where $\lhd \in \{<, <=, ==, >, >=\}$.
(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 1(a). A corresponding function in pure C is given in Figure 1(b). To minimize the extra cost caused by the loop control, the loop is ended conditionally as recommended by Sedgewick [1978]. 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. 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 [Kernighan and Ritchie 1988, 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.

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

(a)

```c
#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)

Fig. 1. (a) A sample program and (b) its translation into pure C.

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 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 [1996]), 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.

2.3 Penalty cost model

We use a weighted version of the cost model presented in [Katajainen and Träff 1997]. Alternatively, one can see the cost model as a simplification of the Fortran-based model discussed in [Saavedra and Smith 1995].

(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, $i \in \{1, 2, \ldots, \ell\}$.

Now, if a program executes n pure-C operations and incurs m_i misses at memory level i, $i \in \{1, 2, \ldots, \ell\}$, 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.

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 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

```
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;
}
```

Fig. 2. Summing integers in a special way.

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 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 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 [1995] 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 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. PRELIMINARY DEFINITIONS

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

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 $\otimes$ is called a **strict weak ordering** if it is irreflexive, transitive, and if the relation $\ominus$ defined by

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

is transitive. Informally, this means that two elements whose relative order is not determined by the relation $\otimes$ 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 $x \otimes y$. The **converse relation** $\oslash$ of relation $\otimes$ is defined by

$$x \oslash y \text{ if and only if } y \otimes x\,.$$

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 [Knuth 1997, 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. 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"
```

Fig. 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 $\otimes$ 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., $x \otimes y$ 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.

3.2 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 [Stroustrup 1997, 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 4.2) with the sift-hole-down-sift-up heuristic [Knuth 1998, Exercise 5.2.3.18], which saves some comparisons [McDiarmid and Reed 1989], but in practice the heap construction becomes slower due to an increase in the number of element moves.

4. KNOWN METHODS AND THEIR ANALYSIS

In this section, we recall and analyse meticulously the heap-construction methods presented by Williams [1964], Floyd [1964], and Fadel et al. [1999]. 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.

4.1 Repeated insertion

In the repeated-insertion heap-construction method proposed by Williams [1964], 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 4 a C++ implementation of the sift-up function and in Figure 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.

In the worst case, each level of the heap is visited exactly once in the inner loop

```
// 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;
  }
}
```

Fig. 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;
  }
}
```

Fig. 5. Williams' heap-construction program in C++.

(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., [Wegener 1992].) 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 [1991] 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 [1985] 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 1. [Sleator and Tarjan 1985] *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 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 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 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 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

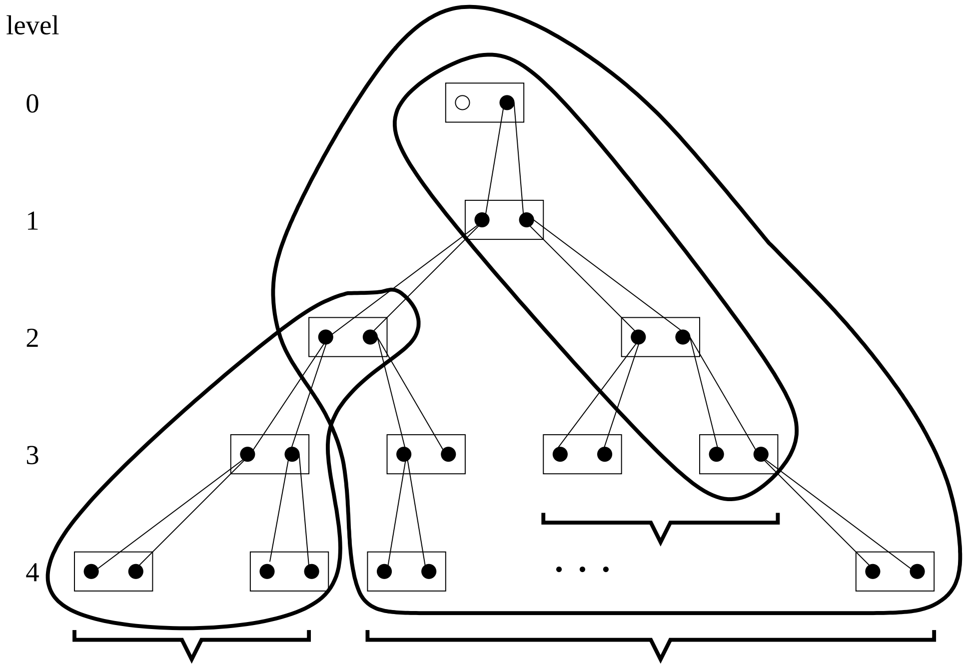

Fig. 6. Illustration of the proof of the lower bound for $B = 2$, $M = 8$, and $N = 31$. The blocks inside the cache after processing the last $M/2$ elements at level 3, those after processing the first group of elements at level 4, and those fetched in when processing the second group of elements at level 4 are circled.

the 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 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

```
// 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;
  }
}
```

Fig. 7. The sift-down function in C++.

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 $(3/2)N/B - M/B - 1$. □

4.2 Repeated merging

In the repeated-merging heap-construction method proposed by Floyd [1964] 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 7 gives a C++ implementation of the sift-down function and Figure 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);
  }
}
```

Fig. 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 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, $1 \leq h < \log_2 B$. 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 $2^{t-1} \geq 2M$, 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, $1 \leq h < \log_2 B$. That is, for each h, $1 \leq h < \log_2 B$, 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. $\square$

4.3 Layerwise construction

In the layerwise heap-construction method proposed, e. g., by Fadel et al. [1999] 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 $\olessthan$ 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* $\ogreaterthan$ 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 [1961], which is randomized and which executes a linear number of instructions in the average case; and the deterministic algorithm of Blum et al. [1973] 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 [Horowitz et al. 1998, 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 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.

(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.

```
// 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;
    }
  }
}
```

Fig. 9. Layerwise heap-construction method in C++.

(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 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., [Raman 1994]) 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 expected number of element exchanges performed is less than $3.39N/6$ as shown by Sedgewick [1977]. By inspecting the program, the number of memory reads is seen

```
#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 { ... }
}
```

Fig. 10. The partition function in C++.

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. [1999] 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. *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 randomized 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 [Horowitz et al. 1998, Section 3.6] and the 3-way partitioning program described in [Wegner 1985]. 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 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 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 $2N/B - 2M/B - O(\log_2 N)$. $\square$

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 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 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.

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 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 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 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.*

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.*

```
// 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);
    }
}
```

Fig. 11. Reengineered version of the repeated-insertion heap-construction method in C++.

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 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 $f_{OPT}(s) \leq f_A(s)$, so we can use Lemma 1 to conclude that, for a real number $r > 1$, $f_{LRU}(s) \leq (r/(r-1))N/B$ if $m_{LRU} = M/B \geq r\lfloor \log_2 N \rfloor$.

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. $\square$

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

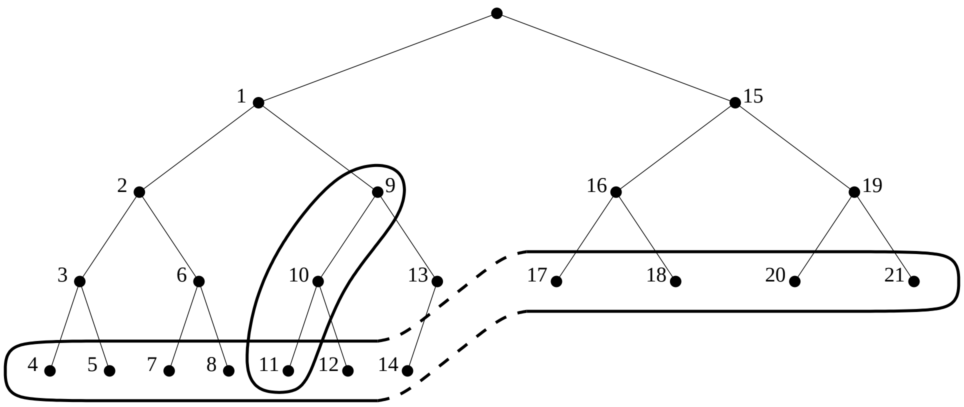

Fig. 12. The depth-first ordering of the sift-up operations. The first loop traverses through the leaves from left to right with a break at the last leaf; the second loop traverses the tree upwards from a leaf that is a left child to the first ancestor that is a right child, performing the sift-up operation at each node on the way down; and the third loop is the inner loop in the sift-up operation itself.

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 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 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 [Knuth 1998, Exercise 5.2.1.33].

In addition to these improvements specific to this heap-construction program, we

```
#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);                                                \
  }
```

Fig. 13. Macros for sifting up one or more elements.

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 [1978].

The resulting program is given in two parts in Figures 13 and 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

```
// 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)
    }
}
```

Fig. 14. Tuned version of the reengineered repeated-insertion heap-construction method in C**++**.

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.

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 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 15.

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

Theorem 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 $M/B \gg \log_2(M/B)$.

```
// 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);
    }
}
```

Fig. 15. Reengineered version of the repeated-merging heap-construction method in C++.

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.

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. □

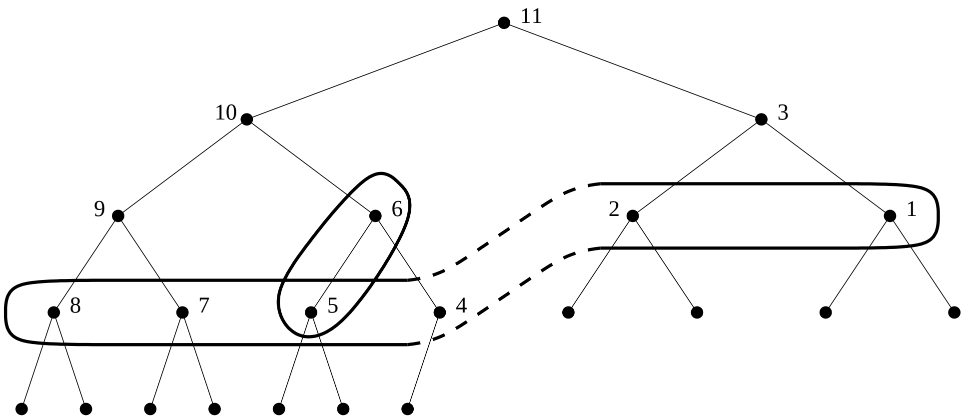

Fig. 16. The depth-first ordering of the sift-down operations. The first loop traverses through the branches of height one from right to left with a break at the last branch; the second loop traverses the tree upwards from a branch that is a left child to the first ancestor that is a right child, performing 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.

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 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

```
#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);                           \
}
```

Fig. 17. Definition of the sift-down operation as a macro.

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));
}
```

```
#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);
  }
}
```

Fig. 18. Tuned version of the reengineered repeated-merging heap-construction method in C++.

The resulting macro definitions for the sift-down operation are given in Figure 17, and the tuned version of the repeated-merging heap-construction method in Figure 18. To help understanding the tuned sift-down macro in Figure 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 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 $10N/4 + 9(3/4)N + 6N/4 + 5N/4 + 6N/4 + O(\log_2 N) = 13.5N + O(\log_2 N)$.

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 [1999] in an external-memory setting. We have followed the first alternative by implementing the random-sampling selection method of Floyd and Rivest [1975] in the form proposed by Sibeyn [1999]. 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	$\mid Y$-section $\mid$	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 [1985] 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 $\mid$ X-section $\mid$ $?_1$	$?_2$ $\mid$ Z-section $\mid$ 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 [1999] 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 [1999] 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 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 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. □

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 [Spork 1999]. 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 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 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 4 and 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 19 and 20 (ascending input), 21 and 22 (descending input), and 23 and 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 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 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 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 [1996].

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 4.3 and 5.3, but our preliminary implementations were not competitive. At the moment we are willing to follow Sedgewick's recommendation [Sedgewick 1977] not to add any overhead to deal with equal elements.

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

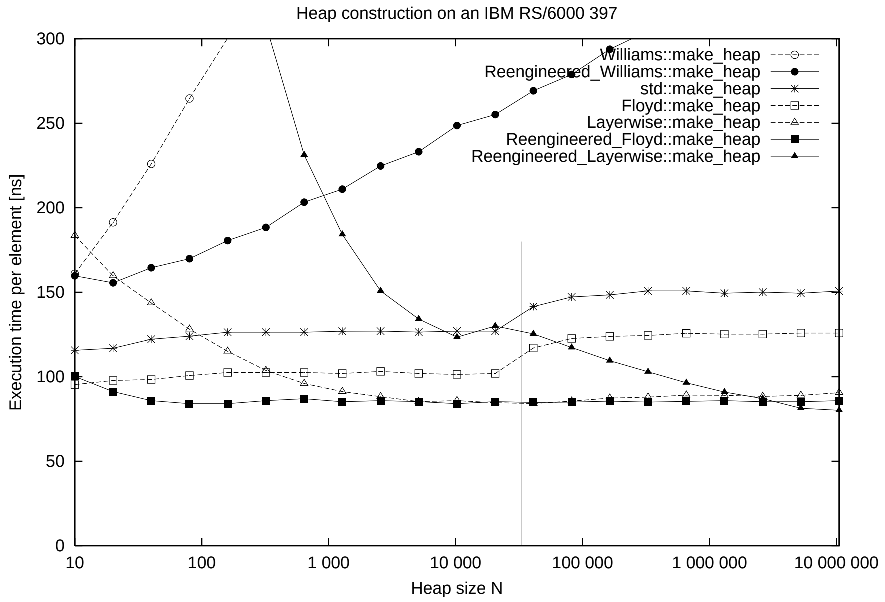

Fig. 19. Execution times of various heap-construction programs on an IBM RS/6000 for an ascending sequence $\{0, 1, \ldots, N-1\}$.

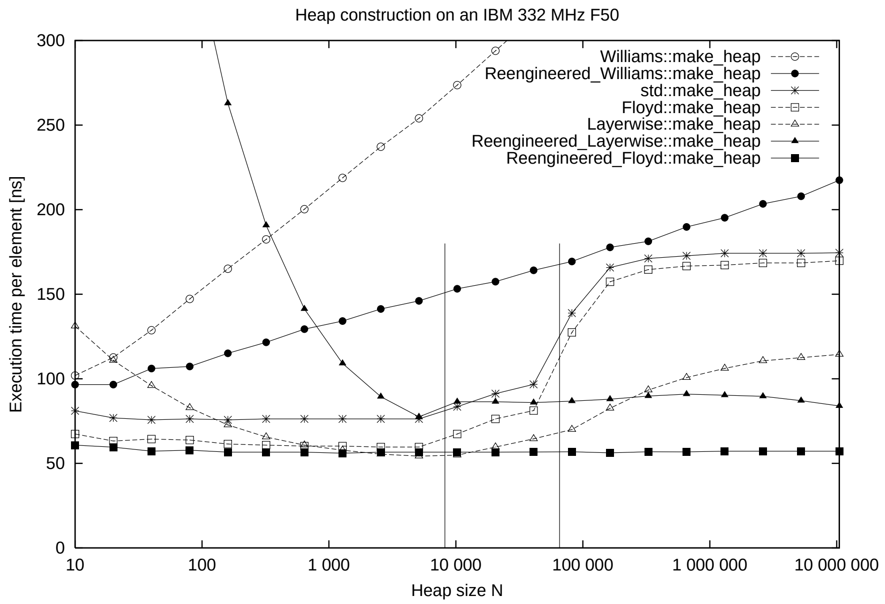

Fig. 20. Execution times of various heap-construction programs on an IBM F50 for an ascending sequence $\{0, 1, \ldots, N-1\}$.

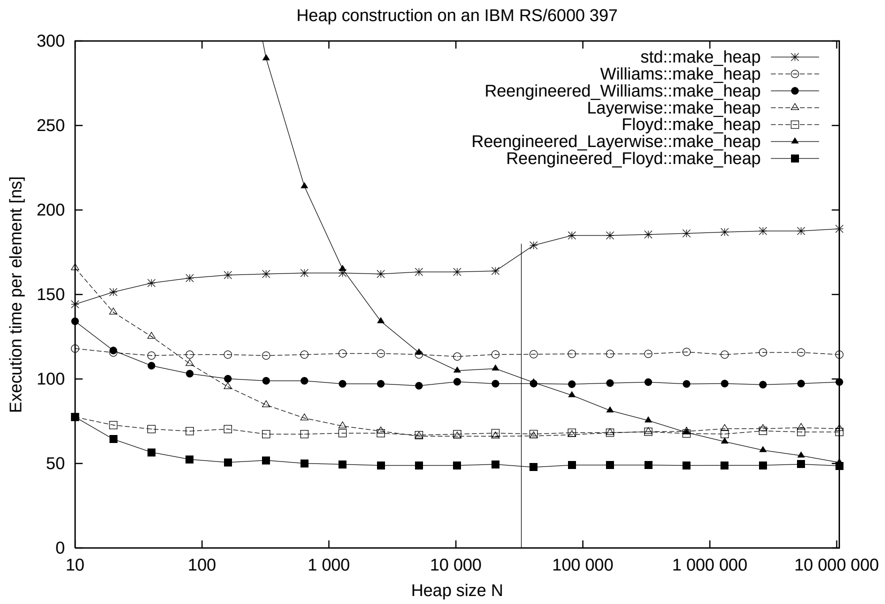

Fig. 21. Execution times of various heap-construction programs on an IBM RS/6000 for a descending sequence $\{N-1, N-2, \ldots, 1, 0\}$.

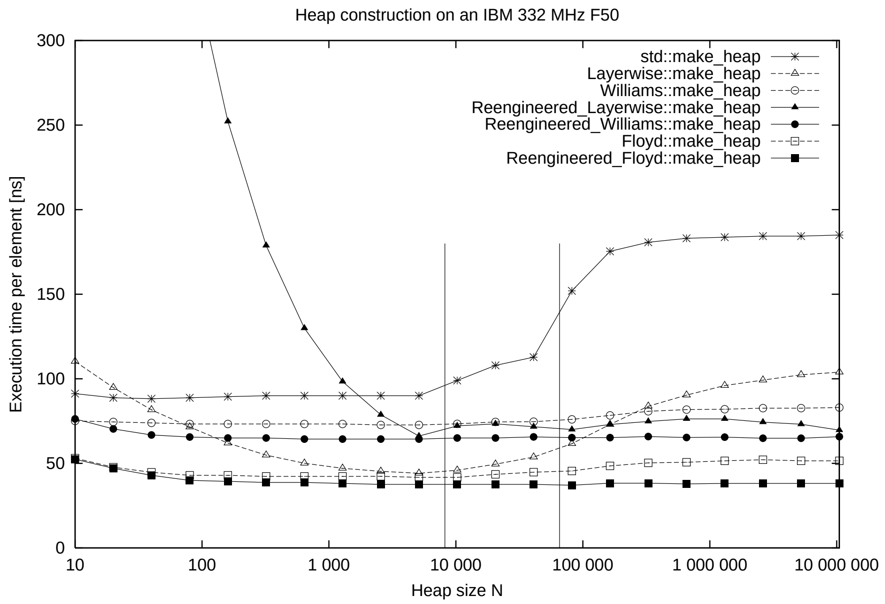

Fig. 22. Execution times of various heap-construction programs on an IBM F50 for a descending sequence $\{N-1, N-2, \ldots, 1, 0\}$.

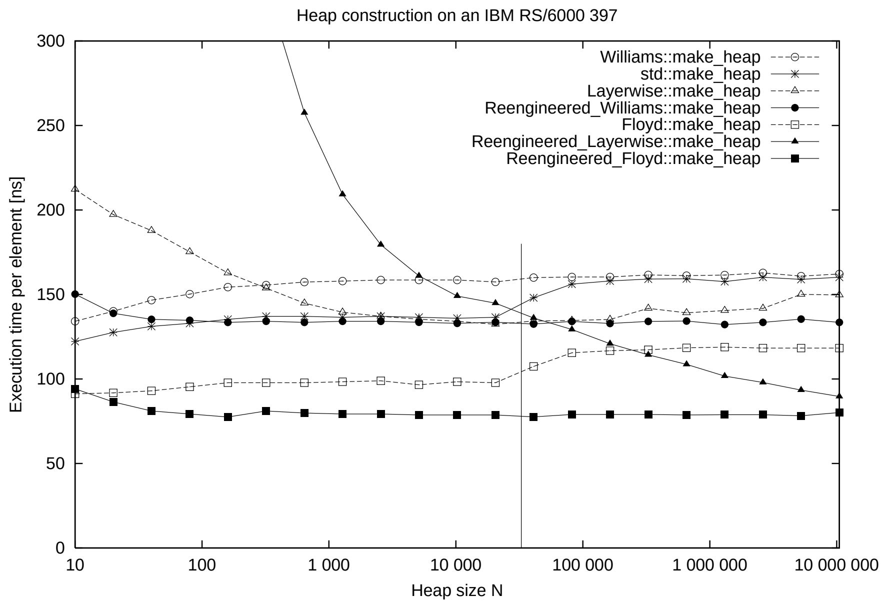

Fig. 23. Execution times of various heap-construction programs on an IBM RS/6000 for a random permutation of $\{0, 1, \ldots, N-1\}$.

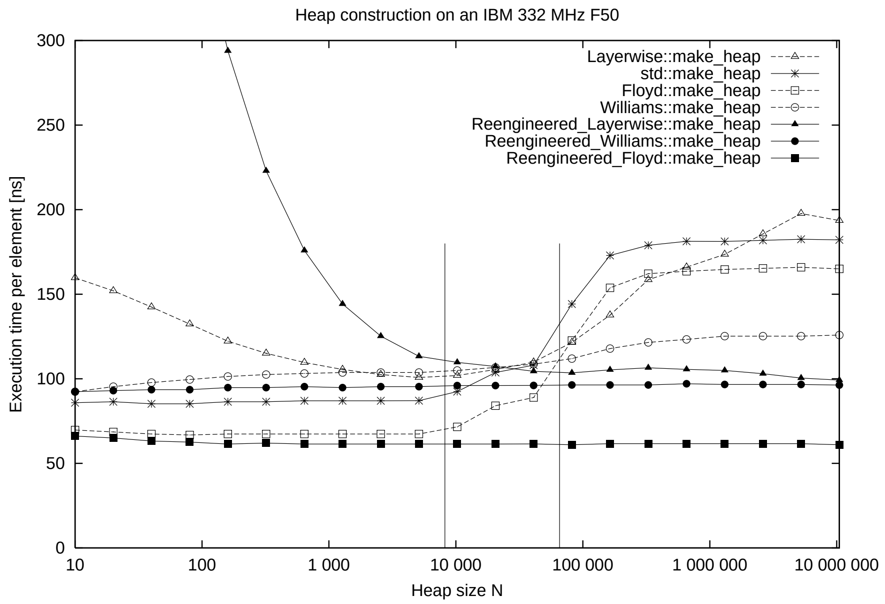

Fig. 24. Execution times of various heap-construction programs on an IBM F50 for a random permutation of $\{0, 1, \ldots, N-1\}$.

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 [Knuth 1997; Knuth 1998]. 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 know to perform well even when compared to the optimal replacement strategy, as shown by Sleator and Tarjan [1985]. 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 [1996].

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

BLUM, M., FLOYD, R. W., PRATT, V., RIVEST, R. L., AND TARJAN, R. E. 1973. Time bounds for selection. *Journal of Computer and System Sciences 7*, 448–461.

CARLSSON, S. 1992. A note on HEAPSORT. *The Computer Journal 35*, 410–411.

FADEL, R., JAKOBSEN, K. V., KATAJAINEN, J., AND TEUHOLA, J. 1999. Heaps and heapsort on secondary storage. *Theoretical Computer Science 220*, 345–362.

FLOYD, R. W. 1964. Algorithm 245: TREESORT 3. *Communications of the ACM 7*, 701.

FLOYD, R. W. AND RIVEST, R. L. 1975. Expected time bounds for selection. *Communications of the ACM 18*, 165–172.

HAGERUP, T. 1998. 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* (1998), pp. 366–398. Springer-Verlag, Berlin/Heidelberg.

HAYWARD, R. AND MCDIARMID, C. 1991. Average case analysis of heap building by repeated insertion. *Journal of Algorithms 12*, 126–153.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach* (2nd ed.). Morgan Kaufmann Publishers, Inc., San Francisco.

HOARE, C. A. R. 1961. Algorithm 65: FIND. *Communications of the ACM 4*, 321–322.

HOROWITZ, E., SAHNI, S., AND RAJASEKARAN, S. 1998. *Computer Algorithms.* Computer Science Press, New York.

KATAJAINEN, J. AND TRÄFF, J. L. 1997. 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* (1997), pp. 217–228. Springer-Verlag, Berlin/Heidelberg.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language* (2nd ed.). Prentice Hall, Englewood Cliffs.

KNUTH, D. E. 1997. *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison Wesley Longman, Reading.

KNUTH, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison Wesley Longman, Reading.

LAMARCA, A. AND LADNER, R. E. 1996. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics 1*, Article 4.

LAMARCA, A. AND LADNER, R. E. 1999. The influence of caches on the performance of sorting. *Journal of Algorithms 31*, 66–104.

MCDIARMID, C. J. H. AND REED, B. A. 1989. Building heaps fast. *Journal of Algorithms 10*, 352–365.

MORET, B. M. E. AND SHAPIRO, H. D. 1991. *Algorithms from P to NP, Volume 1: Design & Efficiency.* The Benjamin/Cummings Publishing Company, Inc., Redwood City.

RAMAN, R. 1994. A simpler analysis of Algorithm 65 (Find). *SIGACT News 25*, 86–89.

SAAVEDRA, R. H. AND SMITH, A. J. 1995. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computers 44*, 1223–1235.

SANDERS, P. 1999. 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* (1999), pp. 312–327. Springer-Verlag, Berlin/Heidelberg.

SEDGEWICK, R. 1977. The analysis of Quicksort programs. *Acta Informatica 7*, 327–355.

SEDGEWICK, R. 1978. Implementing Quicksort programs. *Communications of the ACM 21*, 847–857. Corrigendum, *ibidem 22* (1979), 368.

SIBEYN, J. F. 1999. External selection. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science*, Volume 1563 of *Lecture Notes in Computer Science* (1999), pp. 291–301. Springer-Verlag, Berlin/Heidelberg.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 202–208.

SPORK, M. 1999. Design and analysis of cache-conscious programs. Master's thesis, Department of Computing, University of Copenhagen, Copenhagen. Available from `http://www.diku.dk/research-groups/performance_engineering/`.

STROUSTRUP, B. 1997. *The C++ Programming Language* (3rd ed.). Addison-Wesley, Reading.

WEGENER, I. 1992. 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.

WEGNER, L. M. 1985. Quicksort for equal keys. *IEEE Transactions on Computers C-34*, 362–367.

WILLIAMS, J. W. J. 1964. Algorithm 232: HEAPSORT. *Communications of the ACM 7*, 347–348.