Performance engineering case study: heap construction

Jesper Bojesen[1], Jyrki Katajainen[2], and Maz Spork[2]

[1] UNI•C, Danish Computing Centre for Research and Education, Technical University of Denmark, Building 304, DK-2800 Lyngby, Denmark
[2] Department of Computing, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark

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

1 Introduction

In many research papers on sorting problems, the performance of competing methods is compared by analysing the number of key comparisons, after which it is stated that the number of other operations involved is proportional to that of key comparisons. However, in the case of small keys a key comparison can be as expensive as, for instance, an index comparison. Moreover, to save some key comparisons complicated index manipulation might be necessary, and as the result the methods tuned the number of key comparisons in mind are often impractical (cf., [1]).

Another tradition popularized by Knuth (see, e. g., [9]) 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 to the slowest: register file, on-chip cache, off-chip cache, main memory, and disk storage. The processors have only access to the lowest memory level, the register file, so the data has to be there before it can be processed. The basic idea is to keep the regions of recently accessed memory as close to the processors as possible. If the memory reference cannot be satisfied by one memory level, i. e., in the case of a *miss* or a *fault*, the data must be fetched from the next level. Not only one datum is fetched, but a *block* of data in the hope that the memory locations in vicinity of the current one will be accessed soon. 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 [20] in a hierarchical memory system. Especially, we consider the highest memory level that is too small to fit the whole heap. We call that particular level simply the *cache*. It should be, however, emphasized that our analysis is also valid for the memory levels below this cache, 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., [10, 15]) and heapsort based on a binary heap is not the fastest sorting method (see, e. g., [11, 13]), 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 [20] and Floyd [3] in 1964. In our preliminary experiments we translated these old programs to 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 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 [5], which is guaranteed in the worst case by Floyd's program.
2. Williams' program accesses the memory much more locally than Floyd's program, as shown experimentally by LaMarca and Ladner [10].

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 $(4r/(r-1))N/B$ cache misses. On the other hand, there exists an input which makes Floyd's program to incur more than $((1/2)N \log_2 B)/B$ cache misses if $M \leq (1/4)N$. However, by memory tuning the number of cache misses incurred can be reduced close to the optimal bound N/B if $M \gg 2B^2$.

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 [15]. A fast external-memory algorithm for heap construction have been presented by Fadel et al. [2]. 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.

In this conference version of the paper the proofs of the theorems are omitted. These will be brought out in the full version of the paper. A more thorough experimental evaluation of the methods will also be presented in the full version.

2 Meticulous analysis

In meticulous analysis, the term coined by Katajainen and Träff [8], the goal is to analyse the running time of a program as exactly as possibly, including the constant factors. This style of analysis is well-known from Knuth's books (see, e. g., [9]). 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., [4]) 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 the 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++ [19] and compiled to pure C [8]. Pure C statements are similar in strength to machine instructions on a typical present-day load-store reduced-instruction-set computer (RISC). Let `x`, `y`, and `z` be symbolic names of arbitrary (not necessarily distinct) registers, `c` a constant, `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 not 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 = x`".
3. *Constant-to-register assignments*: "`x = c`".
4. *Register-to-register assignments*: "`x = y`" .
5. *Unary arithmetic assignments*: "`x =` $\ominus$`y`", where $\ominus \in$ {`-`, `~`, `!`}.
6. *Binary arithmetic assignments*: "`x = y` $\oplus$ `z`", where $\oplus \in$ {`+`, `-`, `*`, `/`, `&`, `|`, `^`, `<<`, `>>`}.

7. *Conditional branches*: "`if (x` $\triangleleft$ `y) goto` λ", where $\triangleleft \in \{$`<`, `<=`, `==`, `>`, `>=`$\}$.
8. *Unconditional branches*: "`goto` λ".

2.2 Hierarchical memory model

Let w denote the *word size* of the word RAM. 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 is not present at some level, i. e., when a *miss* occurs, it is fetched from the next 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. 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 [6]), 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 [8]. Alternatively, one can see the cost model as a simplification of the Fortran-based model discussed in [14].

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]` to 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 to 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 misses.

3 Heap construction

An array $\mathbf{a}[1..N]$ of N elements, each with an associated key, is a *binary maximum heap*, or briefly a *heap*, if the key of $\mathbf{a}[\lfloor j/2 \rfloor]$ is larger than or equal to that of $\mathbf{a}[j]$ for all $j \in \{2, 3, \dots, N\}$. We consider the problem of transforming an arbitrary array into a heap. If possible, we would like to carry out the heap construction *in-place*, i. e., by using no more than $O(1)$ extra words of memory.

One can view a heap as a binary tree in which node $\mathbf{a}[1]$ is the *root*, the *children* of node $\mathbf{a}[i]$ are $\mathbf{a}[2i]$ and $\mathbf{a}[2i+1]$ (if they exist), and the *parent* of node $\mathbf{a}[i]$ is $\mathbf{a}[\lfloor i/2 \rfloor]$ (if this exists). The *depth* or *level* of a *node* in a heap 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 heap is the length of the longest path from that node to a node with no children. The *height* of a *heap* is the maximum height (or depth) of any of its nodes. That is, the height of a heap of size N is $\lfloor \log_2 N \rfloor$, and this is also the maximum depth of any of the nodes.

For the purpose of our analysis, we assume that the heap is B*-aligned*, i. e., that there is a block boundary between the root $\mathbf{a}[1]$ and its first child $\mathbf{a}[2]$. That is, the root is in its own block. 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. Recall also that we assumed B to a power of 2. These assumptions guarantee that the children of a node will always be situated in the same block.

In this section we recall and analyse meticulously the heap-construction methods presented by Williams [20], Floyd [3], and Fadel et al. [2]. Both the number of instructions performed and the number of cache misses incurred are analysed. The instruction counts are measured in pure C operations under the assumption that the elements are one-word integers.

3.1 STL compatibility

We have implemented our programs in C++ in the form compatible with the `make_heap` routine available in the Standard Template Library (STL). This routine has two overloaded interfaces:

```
template <class random_access_iterator>
void make_heap(random_access_iterator first, random_access_iterator last);

template <class random_access_iterator, class compare>
void make_heap(random_access_iterator first, random_access_iterator last, compare less);
```

The first one uses `operator<` provided for the elements in the comparisons, whereas the second one accepts any function object with `operator()` as its third parameter.

The input sequence is passed by two iterators `first` and `last`, where `first` points to the first element of the sequence and `last` to the first element beyond the sequence. One complication caused by this arrangement is that we only know the type of the iterators, but not the type of the elements referred to by the iterators. To dereference an iterator and manipulate the resulting element, or to operate with the difference of two iterators, we utilize the `iterator_traits` facility [19, Section 19.2.2] provided by the STL.

The `make_heap` routine of the STL (as provided with the GNU `g++` 2.8.1 and Kuch & Associates `kcc` 3.2f compilers) implements Floyd's method (see Section 3.3) with the siftholedown-siftup heuristic [9, Exercise 5.2.3.18], which saves some comparisons [12], but in practice the heap construction becomes slower due to an increase in the number of element moves.

3.2 Repeated insertion

In Williams' heap-construction method [20] one starts with a heap of size one, and then repeatedly inserts the other elements, one by one, into the heap constructed so far. In Figure 1 a C++ implementation of this method, very much in the spirit of Williams' original program, is given.

In the worst case the inner loop in the `Williams::inheap` routine visits each level of the heap exactly once. Since the height of a heap of size j is $\lfloor \log_2 j \rfloor$, the construction of an N-element heap requires $\sum_{j=2}^{N} \lfloor \log_2 j \rfloor$ key comparisons, which is at most $N \log_2 N - 1.91 \ldots N + O(\log_2 N)$ key comparisons. By simple transformations the inner loop is seen to execute 9 pure C operations, and the outer loop (after inlining `Williams::inheap`) 7 pure C operations. Hence, the pure C operation count is at most $9N \log_2 N - 9N + O(\log_2 N)$.

Hayward and McDiarmid [5] proved that, under the assumption that each initial permutation of the given N elements with distinct keys is equally likely, the number of promotions performed, i. e., assignments `a[i] = a[j]` 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 $24N$.

Let us now analyse the cache behaviour of the repeated-insertion heap-construction method. We say that a block is *relevant* if it contains a node of the heap that is on the path from the current node, indicated by variable `j` in `Williams::make_heap`, to the root. Further, we call a block *active* if it is in the cache, and *inactive* otherwise. When the elements at one level are inserted into the heap, each of the blocks holding a node from any of the previous levels will be relevant only once. Assuming that the capacity of the cache were at least

```
namespace Williams {
  #include <iterator>

  template<class node, class distance, class element, class compare>
  void inheap(node a, distance& n, element in, compare less) {
    iterator_traits<node>::difference_type i, j;
    i = n = n + 1;
  scan:
    if (i > 1) {
      j = i / 2;
      if (less(a[j], in)) {
        a[i] = a[j];
        i = j;
        goto scan;
      }
    }
    a[i] = in;
  }

  template<class node, class compare>
  void make_heap(node first, node last, compare less) {
    iterator_traits<node>::difference_type n = last - first;
    if (n < 2) return;
    node a = first - 1;
    iterator_traits<node>::difference_type j;

    j = 1;
  L:
    inheap(a, j, a[j + 1], less);
    if (j < n) goto L;
  }
}
```

Fig. 1. Williams' repeated-insertion heap-construction method in C++.

$\lfloor \log_2 N \rfloor + 1$ and that we could keep the relevant blocks active all the time, each block holding a node of the heap would be visited at most once.

The ith level of an N-element heap, $i \in \{0, 1, \ldots, \lfloor \log_2 N \rfloor\}$, contains at most 2^i elements which occupy at most $\lceil 2^i/B \rceil + 1$ blocks. Hence, the insertion of these 2^i elements into the heap can incur at most $\sum_{j=0}^{i}(\lceil 2^j/B \rceil + 1)$ cache misses. Summing this over all possible levels gives

$$\sum_{i=0}^{\lfloor \log_2 N \rfloor} \sum_{j=0}^{i} (\lceil 2^j/B \rceil + 1)$$

cache misses, which is at most $4N/B + O\left((\log_2 N)^2\right)$.

The main problem with the above analysis is that we cannot control the replacement of the cache blocks, but this is done by the cache system according to the LRU replacement strategy. In particular, it may be possible that a relevant block is removed from the cache and it has to be reread into the cache if it is needed later on. This complication is formally handled in the proof of the following theorem. The main ingredient of the proof is a result by Sleator and Tarjan [17, Theorem 6] which shows that the LRU strategy cannot be much worse than the optimal replacement strategy if the size of the cache is a bit larger than that used by the optimal algorithm.

Theorem 1. *Let B denote the size of cache blocks and N the size of the heap to be constructed. Assuming that the cache can hold* at least $r\lfloor\log_2 N\rfloor$ *blocks for some real number $r > 1$ and that the block-replacement strategy is LRU, the repeated-insertion heap-construction method incurs at most $(4r/(r-1))N/B + O\left((\log_2 N)^2\right)$ cache misses.*

Our simulations show that the upper bound given by Theorem 1 is pessimistic. On the other hand, even in the best case Williams' method will touch two levels of the heap at each insertion. Based on this observation it is not difficult to show that $(3/2)N/B - M/B$ cache misses will always be necessary, so the number of misses cannot be made arbitrary close to the optimal bound N/B if $N \gg M$.

The worst-case performance of Williams' heap-construction method is dominated by the internal computations. Hence, it is motivated to try to reduce the number of instructions performed. The code tuning of the program should, however, be accomplished carefully since on an average both the inner loop and the outer loop are executed about equally many times. Hence, by 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 code tuning we used the following principles:

1. All common subexpressions were removed.
2. The function `Williams::inheap` was inlined.
3. Instead of array indexing we used the pointer notation provided by the C++ language. Sometimes inelegant casts were necessary to avoid the extra multiplications and divisions that were generated by a compiler when calculating a difference of two iterators.
4. Before inserting an element into the heap we compared its key to the key of 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 with a key larger than or equal to the key of the inserted element found.
5. To compensate the extra key comparison the root was kept in registers.
6. By making two copies of both of the two versions of the inner loop the assignment `i = j` could be avoided as hinted, e. g., in [9, Exercise 5.2.1.33].
7. The loops were finished conditionally as recommended by Sedgewick [16]. This made the program code complicated since now the outer loop is ended in four different places.

Because of space restrictions we cannot give the resulting program here. Note, however, that the foregoing transformations do not change the memory reference pattern of the program.

After code tuning each of the copies of the inner loop executes 5 pure C operations and the copies of the outer loop 8 or 11 operations depending on which of the branches is selected. For an average-case input the longer branch is selected almost always [5], whereas for a worst-case input (a sorted array) the shorter branch is executed at all times. From this we get that the number of pure C operations executed is bounded by $(5 \cdot 1.3 + 11)N + O(\log_2 N) =$

```
namespace Floyd {
  #include <iterator>

  template<class node, class distance, class compare>
  void siftdown(node a, distance i, distance n, compare less) {
    iterator_traits<node>::value_type copy;
    iterator_traits<node>::difference_type  j;

    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;
  }

  template <class node, class compare>
  void make_heap(node first, node last, compare less) {
    node a = first - 1;
    iterator_traits<node>::difference_type n = last - first;
    iterator_traits<node>::difference_type i;

    for (i = n / 2; i > 0; i--)
      siftdown(a, i, n, less);
  }
}
```

Fig. 2. Floyd's repeated-merging heap-construction method in C++.

$17.5N + O(\log_2 N)$ in the average case and by $5N \log_2 N + 8N + O(1)$ in the worst case.

3.3 Repeated merging

In Floyd's heap-construction method [3] small heaps are merged to form larger heaps. The basic operation (`Floyd::siftdown`) transforms a subtree rooted by a node containing an arbitrary element and the heaps rooted by the children of that node into a heap. In Floyd's scheme the subtrees are processed in order of their heights. Trivially, all subtrees with height 0 form a heap. After this the subtrees with height 1 are processed, then the subtrees with height 2 and so on. Figure 2 gives a C++ program, similar to Floyd's original program, that implements this heap-construction method.

To make the program comparable with the tuned version of Williams' program, we tuned it by using the same principles as earlier with two substantial additions. First, the test of the special case, whether a node has only one child, was moved outside the inner loop. Second, a specialized version of siftdown was written which handles a heap of exactly three elements. Since roughly half of the calls to siftdown are handled by this specialized routine, noticeable savings are achieved. Except for these extra optimizations, we expect that most of our optimizations can be carried out automatically by modern compilers, so the main concern here is to get a realistic estimate for the instruction count of the program. Again due to space restrictions we do not give the resulting program, but note that the transformations do not change the memory reference pattern.

The specialized siftdown is called at least $\lfloor N/4 \rfloor - 1$ times and at most $\lfloor N/4 \rfloor$ times, and it executes at most 12 pure C operations. The control of the outer loop requires 2 additional pure C operations per iteration. It is known that Floyd's method constructs a heap of size N with at most $2N$ key comparisons. Of these at least $N/2 - O(1)$ are performed by the specialized siftdown and the remaining $(3/2)N + O(1)$ by the normal siftdown. Since 2 comparisons are performed in each iteration of the original inner loop, we conclude that the inner loop of the normal siftdown is repeated $(3/4)N + O(1)$ times. All branches of the three copies of the inner loop execute 9 pure C operations, and the outer loop 9 pure C operations. Hence, the total number of pure C operations executed is at most $14(N/4) + 9(3/4)N + 9(N/4) + O(1) = 12.5N + O(1)$.

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

Theorem 2. *Let B denote the size of cache blocks, and assume that the block-replacement strategy is LRU. There exists an input of size $N \geq 2B^2$ that makes the repeated-merging heap-construction method, in the form programmed by Floyd, to incur more than $(1/2)(N \log_2 B)/B$ cache misses if the cache can hold at most $(1/4)N/B$ blocks.*

One could also show that the number of cache misses is upper bounded by $(cN \log_2 B)/B$ for a positive constant c. We leave the proof of this fact as an exercise for the reader.

3.4 Reengineered repeated merging

The main drawback in Floyd's method is the order in which the subtrees are considered. The subtrees 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. It is not difficult to write a recursive program that implements this idea.

Due to the recursion stack the recursive program does not construct a heap in-place, but the recursion can be removed such that only a constant amount of indices are in use. A reengineered iterative version of the recursive program is given in Figure 3. The iterative version handles the nodes at the level next to the bottommost level in a special manner: if a node has at least one child, the two or three element heap rooted by it is constructed; after this the nested loop follows the path towards the root, calling siftdown for the parent node, as long as a node is found which is not a left child.

```
namespace Reengineered_Floyd {
  #include <iterator>

  template <class node, class compare>
  void make_heap(node first, node last, compare less) {
    node a = first - 1;
    iterator_traits<node>::difference_type n = last - first;
    iterator_traits<node>::difference_type i, j, k, z;

    for (i = 1; i <= n; i *= 2);

    i = i / 2 - 1, j = i / 2, k = n / 2;
    for (; i > j; i--) {
      if (i <= k)
        Floyd::siftdown(a, i, n, less);
      for (z = i; (z & 1) == 0; z /= 2)
        Floyd::siftdown(a, z / 2, n, less);
    }
  }
}
```

Fig. 3. Reengineered version of the repeated-merging heap-construction method in C++.

In our actual implementation the optimizations of siftdown described in Section 3.3 were included. Furthermore, two versions of the outer loop were written to avoid the `if`-test in it. After these optimizations the cost of the outer loop is reduced to 3 pure C operations per call to siftdown. As earlier the pure C cost of the specialized siftdown is 12 and it is executed at most $N/4$ times. The pure C cost of the inner loop of the normal siftdown is 9 and it is executed $(3/4)N+O(1)$ times. The 7 pure C operations in the normal siftdown outside its inner loop are executed at most $N/4$ times. Finally, the computation of the largest power of 2 smaller than or equal to N requires $O(\log_2 N)$ pure C operations. To sum up, the total cost is $13N + O(\log_2 N)$.

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

Theorem 3. *Let B denote the size of cache blocks and N the size of the heap to be constructed. Assuming that the cache can hold M elements and $M \geq 2B^2$, and that the block-replacement strategy is LRU, the reengineered repeated-merging heap-construction method incurs never more than $N/B+8(\log_2 M-\log_2 B)N/M$ cache misses, which for $M \gg 2B^2$ is close to N/B.*

3.5 Layerwise construction

In the layerwise heap-construction method proposed, e. g., by Fadel et al. [2] the elements to be moved to the nodes with height 0, with height 1, et cetera are determined repeatedly. In each phase one half of the remaining elements are excluded from consideration. The process involves the selection of the median element and the (3-way) partitioning around the median element. Figure 4 implements this heap-construction method by using the STL routine `nth_element`, which accomplishes both the selection and the partitioning.

```
namespace Layerwise {
  #include <algorithm>

  template <class node, class compare>
  void make_heap(node first, node last, compare less) {
    while (last - first > 1) {
      node middle = first + (last - first) / 2;
      nth_element(first, middle, last, less);
      last = middle;
    }
  }
}
```

Fig. 4. Layerwise heap-construction method in C++.

We wrote several programs, both deterministic and randomized, for selection and partitioning, but none of our programs were able to beat the library routine in speed. Therefore, we only used our own programs for analytical purposes. We would like to point out that most selection routines are recursive so they need a recursion stack for their operation, i. e., they do not run in-place, but of course a recursion stack with a logarithmic size is acceptable in practice. Moreover, both the selection and partitioning routines should be able to handle multiset data, i. e., the input can contain elements with equal keys. Therefore, not all published methods can be used for our purposes.

We have not made any attempts to analyse the instruction count of the layerwise heap-construction program as done for the other heap-construction programs. It is well-known that the instruction count is high for the deterministic selection routines, even though it is linear in the worst case. The instruction count is moderate for the randomized selection method proposed by Hoare [7], it can handle multisets, and it performs partitioning simultaneously with the selection. The `nth_element` routine of the STL (as provided with the GNU `g++` 2.8.1 and Kuch & Associates `kcc` 3.2f compilers) is an implementation of this randomized method, but it switches to the deterministic method in a clever way if the depth of the recursion gets too large.

Fadel et al. [2] 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 4. *Let B denote the size of cache blocks and N the size of the heap to be constructed. Assuming that the block-replacement strategy is LRU, the layerwise heap-construction method 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.*

4 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 [18]. Many of the findings in the theoretical part evolved first after we got a deep understanding of the programs with these tools. Third, a micro benchmark, similar to that described in [14], 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 Section 3 were carried out. In this section we give the test results in the case the input is a random permutation of $\{0, 1, \ldots, N-1\}$ and N ranges from 2^4 to 2^{23}. That is, the elements were 4-byte integers.

Based on 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. Further characteristics of these computers are given in Table 1.

Table 1. Characteristics of the computers on which the tests were carried out.

Computer	Clock rate	On-chip cache	Off-chip cache
IBM RS/6000 397	160 MHz	128 kbytes, 128 bytes per cache line, 4-way set associative	none
IBM F50	332 MHz	32 kbytes, 32 bytes per cache line, 4-way set associative	256 kbytes, 64 bytes per cache line, 8-way set associative

The C++ programs developed were compiled by the Kuch & Associates `kcc` 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. In general, the object code generated by this compiler was about 30 percent faster than the corresponding code produced by the GNU `g++` compiler (version 2.8.1).

All the programs described in Section 3 as well as the heap-construction program from the STL (as provided with the GNU `g++` 2.8.1 and Kuch & Associates `kcc` 3.2f compilers) were included in our tests. Their execution times on the two test computers are given in Figures 5 and 6, respectively. For each value of N, the experiment was repeated $2^{24}/N$ times, each time with a new array. Furthermore, the execution trials were carried out on dedicated hardware, so no cache interference from our sources was expected.

The results on IBM RS/6000 indicate clearly the effectiveness of our code tuning. The analytical pure C costs reflect well the speed-ups obtained. The results on IBM F50 show the effectiveness of our memory tuning. In particular, the reengineered version of Floyd's 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.

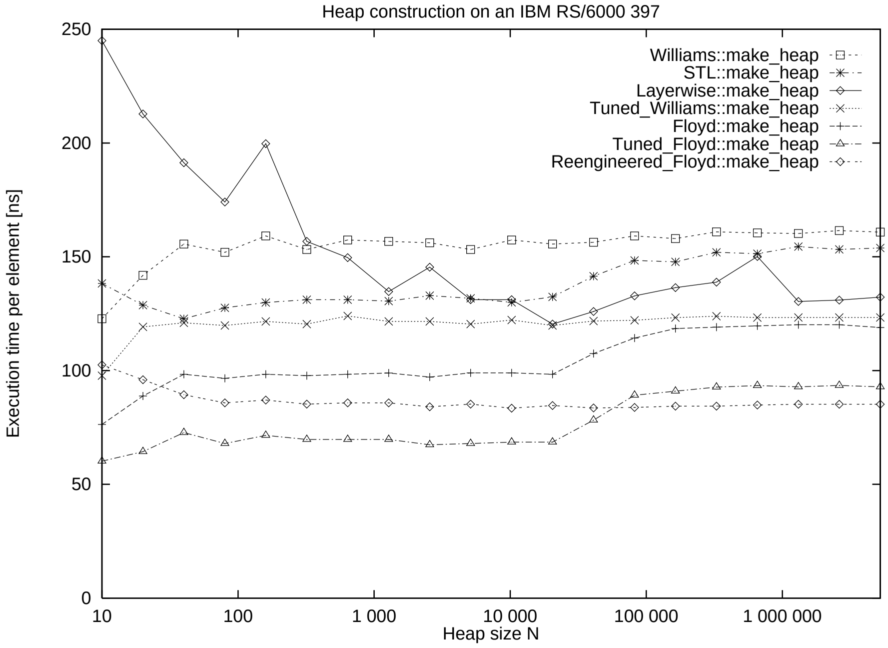

Fig. 5. Execution times of various heap-construction programs on an IBM RS/6000.

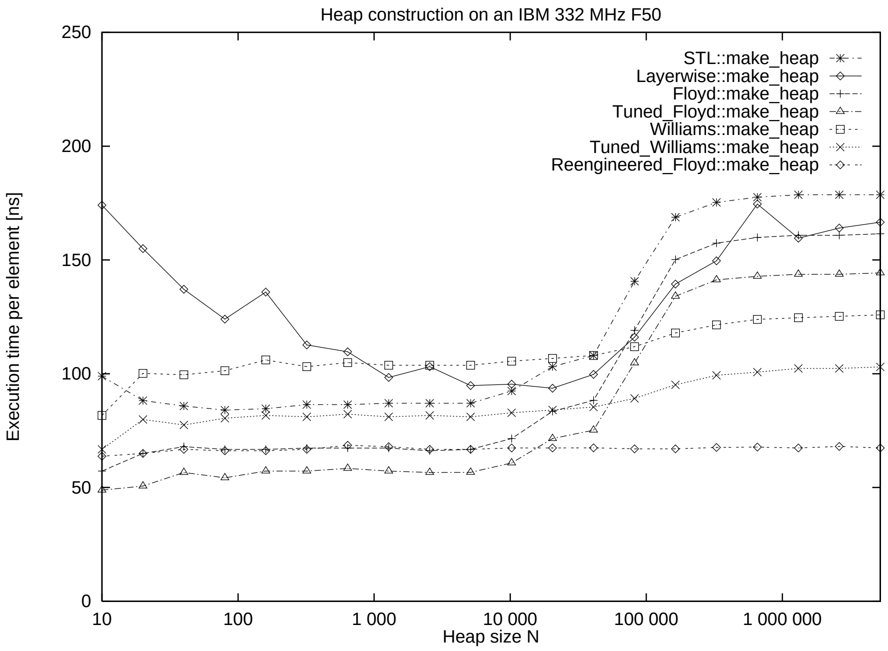

Fig. 6. Execution times of various heap-construction programs on an IBM F50.

References

1. Carlsson, S.: A note on HEAPSORT. *The Computer Journal* **35** (1992), 410–411
2. Fadel, R., Jakobsen, K. V., Katajainen, J., Teuhola, J., Heaps and heapsort on secondary storage, *Theoretical Computer Science* **220** (1999), 345–362
3. Floyd, R. W.: Algorithm 245: TREESORT 3. *Communications of the ACM* **7** (1964), 701
4. Hagerup, T.: Sorting and searching on the word RAM, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **1373**, Springer-Verlag, Berlin/Heidelberg (1998), 366–398
5. Hayward, R., McDiarmid, C.: Average case analysis of heap building by repeated insertion. *Journal of Algorithms* **12** (1991), 126–153
6. Hennessy, J. L., Patterson, D. A.: *Computer Architecture: A Ouantitative Approach*, 2nd Edition. Morgan Kaufmann Publishers, Inc., San Francisco (1996)
7. Hoare, C. A. R.: Algorithm 65: FIND. *Communications of the ACM* **4** (1961), 321–322
8. Katajainen, J., Träff, J. L.: A meticulous analysis of mergesort programs. *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science **1203**, Springer-Verlag, Berlin/Heidelberg (1997), 217–228
9. Knuth, D. E.: *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, 2nd Edition. Addison Wesley Longman, Reading (1998)
10. LaMarca, A., Ladner, R. E.: The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics* **1** (1996), Article 4
11. LaMarca, A., Ladner, R. E.: The influence of caches on the performance of sorting. *Journal of Algorithms* **31** (1999), 66–104
12. McDiarmid, C. J. H., Reed, B. A.: Building heaps fast. *Journal of Algorithms* **10** (1989), 352–365
13. Moret, B. M. E., Shapiro, H. D.: *Algorithms from P to NP*, Volume 1: *Design & Efficiency*. The Benjamin/Cummings Publishing Company, Inc., Redwood City (1991)
14. Saavedra, R. H., Smith, A. J.: Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computers* **44** (1995), 1223–1235
15. Sanders, P.: Fast priority queues for cached memory. *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, Lecture Notes in Computer Science **1619**, Springer-Verlag, Berlin/Heidelberg (1999)
16. Sedgewick, R.: Implementing Quicksort programs. *Communications of the ACM* **21** (1978), 847–857. Corrigendum, *ibidem* **22** (1979), 368
17. Sleator, D. D., Tarjan, R. E.: Amortized efficiency of list update and paging rules. *Communications of the ACM* **28** (1985), 202–208
18. Spork, M.: Design and analysis of cache-conscious programs, M. Sc. Thesis, Department of Computing, University of Copenhagen, Copenhagen (1999). Available from `http://www.diku.dk/research-groups/performance_engineering/publications/spork99.ps.gz`
19. Stroustrup, B.: *The C++ Programming Language*, 3rd Edition. Addison-Wesley, Reading (1997)
20. Williams, J. W. J.: Algorithm 232: HEAPSORT. *Communications of the ACM* **7** (1964), 347–348