Fast Priority Queues for Cached Memory

Peter Sanders

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
sanders@mpi-sb.mpg.de, Fax: (49) 681-9325 199

Abstract. The cache hierarchy prevalent in todays high performance processors has to be taken into account in order to design algorithms which perform well in practice. We advocates the approach to adapt external memory algorithms to this purpose. We exemplify this approach and the practical issues involved by engineering a fast priority queue suited to external memory and cached memory which is based on k-way merging. It improves previous external memory algorithms by constant factors crucial for transferring it to cached memory. Running in the cache hierarchy of a workstation the algorithm is at least two times faster than an optimized implementation of binary heaps and 4-ary heaps for large inputs.

1 Introduction

The mainstream model of computation used by algorithm designers in the last half century [18] assumes a sequential processor with unit memory access cost. However, the mainstream computers sitting on our desktops have increasingly deviated from this model in the last decade [10, 11, 13, 17, 19]. In particular, we usually distinguish at least four levels of memory hierarchy: A file of multi-ported *registers*, can be accessed in parallel in every clock-cycle. The *first-level cache* can still be accessed every one or two clock-cycles but it has only few parallel ports and only achieves the high throughput by pipelining. Therefore, the instruction level parallelism of super-scalar processors works best if most instructions use registers only. Currently, most first-level caches are quite small (8–64KB) in order to be able to keep them on chip and close to the execution unit. The *second-level cache* is considerably larger but also has an order of magnitude higher latency. If it is off-chip, its size is mainly constrained by the high cost of fast static RAM. The *main memory* is build of high density, low cost dynamic RAM. Including all overheads for cache miss, memory latency and translation from logical over virtual to physical memory addresses, a main memory access can be two orders of magnitude slower than a first level cache hit. Most machines have separate caches for data and code so that we can disregard instruction reads as long as the programs remain reasonably short.

Although the technological details are likely to change in the future, physical principles imply that fast memories must be small and are likely to be more

expensive than slower memories so that we will have to live with memory hierarchies when talking about sequential algorithms for large inputs.

The general approach of this paper is to model one cache level and the main memory by the single disk single processor variant of the external memory model [22]. This model assumes an internal memory of size M which can access the external memory by transferring blocks of size B. We use the word pairs "cache line" and "memory block", "cache" and "internal memory", "main memory" and "external memory" and "I/O" and "cache fault" as synonyms if the context does not indicate otherwise. The only formal limitation compared to external memory is that caches have a fixed replacement strategy. In another paper, we show that this has relatively little influence on algorithm of the kind we are considering. Nevertheless, we henceforth use the term *cached memory* in order to make clear that we have a different model.

Despite of the far-reaching analogy between external memory and cached memory, a number of additional differences should be noted: Since the speed gap between caches and main memory is usually smaller than the gap between main memory and disks, we are careful to also analyze the work performed internally. The ratio between main memory size and first level cache size can be much larger than that between disk space and internal memory. Therefore, we will prefer algorithms which use the cache as economically as possible. Finally, we also discuss the remaining levels of the memory hierarchy but only do that informally in order to keep the analysis focussed on the most important aspects.

In Section 2 we present the basic algorithm for our *sequence heaps* data structure for priority queues[1]. The algorithm is then analyzed in Section 3 using the external memory model. For some m in $\Theta(M)$, k in $\Theta(M/B)$, any constant $\gamma > 0$ and $R = \lceil \log_k \frac{I}{m} \rceil \leq \mathcal{O}(M/B)$ it can perform I insertions and up to I `deleteMin`s using $I(2R/B + \mathcal{O}(1/k + (\log k)/m))$ I/Os and $I(\log I + \log R + \log m + \mathcal{O}(1))$ key comparisons. In another paper, we show that similar bounds hold for cached memory with a-way associative caches if k is reduced by $\mathcal{O}(B^{1/a})$. In Section 4 we present refinements which take the other levels of the memory hierarchy into account, ensure almost optimal memory efficiency and where the amortized work performed for an operation depends only on the current queue size rather than the total number of operations. Section 5 discusses an implementation of the algorithm on several architectures and compares the results to other priority queue data structures previously found to be efficient in practice, namely binary heaps and 4-ary heaps.

Related Work

External memory algorithms are a well established branch of algorithmics (e.g. [21, 20]). The external memory heaps of Teuhola and Wegner [23] and the fish-spear data structure [9] need $\Theta(B)$ less I/Os than traditional priority queues like binary heaps. Buffer search trees [1] were the first external memory priority

[1] A data structure for representing a totally ordered set which supports insertion of elements and deletion of the minimal element.

queue to reduce the number of I/Os by another factor of $\Theta(\log \frac{M}{B})$ thus meeting the lower bound of $\mathcal{O}((I/B)\log_{M/B} I/M)$ I/Os for I operations (amortized). But using a full-fledged search tree for implementing priority queues may be considered wasteful. The heap-like data structures by Brodal and Katajainen, Crauser et. al. and Fadel et. al. [3, 7, 8] are more directly geared to priority queues and achieve the same asymptotic bounds, one [3] even per operation and not in an amortized sense. Our sequence heap is very similar. In particular, it can be considered a simplification and reengineering of the "improved array-heap" [7]. However, sequence heaps are more I/O-efficient by a factor of about three (or more) than [1, 3, 7, 8] and need about a factor of two less memory than [1, 7, 8].

2 The Algorithm

Merging k sorted sequences into one sorted sequence (k-way merging) is an I/O efficient subroutine used for sorting – both for external [14] and cached memory [16]. The basic idea of sequence heaps is to adapt k-way merging to the related but more dynamical problem of priority queues.

Let us start with the simple case, that at most km insertions take place where m is the size of a buffer which fits into fast memory. Then the data structure could consist of k sorted sequences of length up to m. We can use k-way merging for deleting a batch of the m smallest elements from k sorted sequences. The next m deletions can then be served from a buffer in constant time.

To allow an arbitrary mix of insertions and deletions, we maintain a separate binary heap of size up to m which holds the recently inserted elements. Deletions have to check whether the smallest element has to come from this *insertion buffer*. When this buffer is full, it is sorted and the resulting sequence becomes one of sequences for the k-way merge.

Up to this point, sequence heaps and the earlier data structures [3, 7, 8] are almost identical. Most differences are related to the question how to handle more than km elements. We cannot increase m beyond M since the insertion heap would not fit into fast memory. We cannot arbitrarily increase k since eventually k-way merging would start to incur cache faults. Sequence heaps use the approach to make room by merging all the k sequences producing a larger sequence of size up to km [3, 7].

Now the question arises how to handle the larger sequences. We adopt the approach used for *improved array-heaps* [7] to employ R *merge groups* $G_1, \ldots, G_R$ where G_i holds up to k sequences of size up to mk^{i-1}. When group G_i overflows, all its sequences are merged and the resulting sequence is put into group G_{i+1}.

Each group is equipped with a *group buffer* of size m to allow batched deletion from the sequences. The smallest elements of these buffers are deleted in batches of size $m' \ll m$. They are stored in the *deletion Buffer*. Fig. 1 summarizes the data structure. We now have enough information to explain how deletion works:

DeleteMin: The smallest elements of the deletion buffer and insertion buffer are compared and the smaller one is deleted and returned. If this empties the deletion buffer, it is refilled from the group buffers using an R-way merge. Before

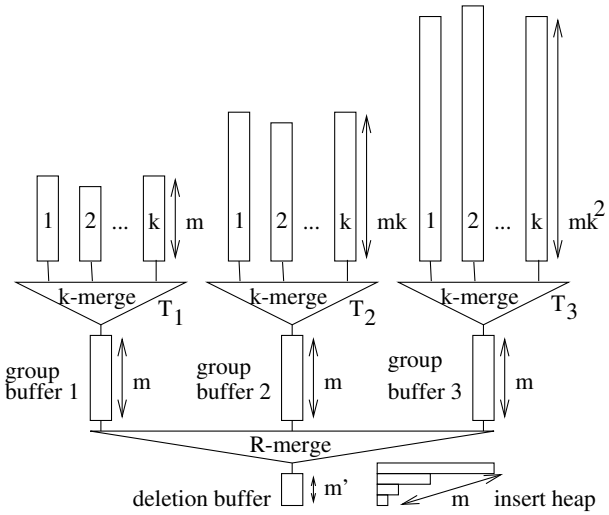

Fig. 1. Overview of the data structure for sequence heaps for $R = 3$ merge groups.

the refill, group buffers with less than m' elements are refilled from the sequences in their group (if the group is nonempty).

DeleteMin works correctly provided the data structure fulfills the heap property, i.e., elements in the group buffers are not smaller than elements in the deletion buffer, and, in turn, elements in a sorted sequence are not smaller than the elements in the respective group buffer. Maintaining this invariant is the main difficulty for implementing insertion:

Insert: New elements are inserted into the insert heap. When its size reaches m its elements are sorted (e.g. using merge sort or heap sort). The result is then merged with the concatenation of the deletion buffer and the group buffer 1. The smallest resulting elements replace the deletion buffer and group buffer 1. The remaining elements form a new sequence of length at most m. The new sequence is finally inserted into a free slot of group G_1. If there is no free slot initially, G_1 is emptied by merging all its sequences into a single sequence of size at most km which is then put into G_2. The same strategy is used recursively to free higher level groups when necessary. When group G_R overflows, R is incremented and a new group is created. When a sequence is moved from one group to the other, the heap property may be violated. Therefore, when G_1 through G_i have been emptied, the group buffers 1 through $i+1$ are merged, and put into G_1.

The latter measure is one of the few differences to the improved array heap [7] where the invariant is maintained by merging the new sequence and the group buffer. This measure almost halves the number of required I/Os.

For cached memory, where the speed of internal computation matters, it s also crucial how to implement the operation of k-way merging. We propose to use the "loser tree" variant of the selection tree data structure described by Knuth [14, Section 5.4.1]: When there are k' nonempty sequences, it consists of a binary tree with k' leaves. Leaf i stores a pointer to the current element of sequence i. The current keys of each sequence perform a tournament. The winner is passed up the tree and the key of the loser and the index of its leaf are stored in the inner node. The overall winner is stored in an additional node above the root. Using this data structure, the smallest element can be identified and replaced by the next element in its sequence using $\lceil \log k \rceil$ comparisons. This is less than the heap of size k assumed in [7, 8] would require. The address calculations and memory references are similar to those needed for binary heaps with the noteworthy difference that the memory locations accessed in the loser tree are predictable which is not the case when deleting from a binary heap. The instruction scheduler of the compiler can place these accesses well before the data is needed thus avoiding pipeline stalls, in particular if combined with loop unrolling.

3 Analysis

We start with an analysis for of the number of I/Os in terms of B, the parameters k, m and m' and an arbitrary sequence of `insert` and `deleteMin` operations with I insertions and up to I `deleteMin`s. We continue with the number of key comparisons as a measure of internal work and then discuss how k, m and m' should be chosen for external memory and cached memory respectively. Adaptions for memory efficiency and many accesses to relatively small queues are postponed to Section 4.

We need the following observation on the minimum intervals between tree emptying operations in several places:

Lemma 1. *Group G_i can overflow at most every $m(k^i - 1)$ insertions.*

Proof. The only complication is the slot in group G_1 used for invalid group buffers. Nevertheless, when groups G_1 through G_i contain k sequences each, this can only happen if $\sum_{j=1}^{R} m(k-1)k^{j-1}So = m(k^i - 1)$ insertions have taken place. ∎

In particular, since there is room for m insertions in the insertion buffer, there is a very simple upper bound for the number of groups needed:

Corollary 1. $R = \lceil \log_k \frac{I}{m} \rceil$ *groups suffice.*

We analyze the number of I/Os based on the assumption that the following information is kept in internal memory: The insert heap; the deletion buffer; a merge buffer of size m; group buffers 1 and R; the loser tree data for groups G_R, G_{R-1} (we assume that $k(B+2)$ units of memory suffice to store the blocks of the k sequences which are currently accessed and the loser tree information

itself); a corresponding amount of space shared by the remaining $R-2$ groups and data for merging the R group buffers.[2]

Theorem 1. *If* $R = \lceil \log_k(I/m) \rceil$, $4m+m'+(3k+R)(B+2) < M$ *and* $k(B+2) \leq m - m'$ *then*

$$I\left(\frac{2R}{B} + \mathcal{O}\left(\frac{1}{k} + \frac{\log k}{m}\right)\right)$$

I/Os suffice to perform any sequence of I `insert`*s and up to* I `deleteMin`*s on a sequence heap.*

Proof. Let us first consider the I/Os performed for an element moving on the following *canonical* data path: It is first inserted into the insert buffer and then written to a sequence in group G_1 in a batched manner, i.e, we charge $1/B$ I/Os to the insertion of this element. Then it is involved in emptying groups until it arrives in group G_R. For each emptying operation it is involved into one batched read and one batched write, i.e., we charge $2(R-1)/B$ I/Os for tree emptying operations. Eventually, it is read into group buffer R. We charge $1/B$ I/Os for this. All in all, we get a charge of $2R/B$ I/Os for each insertion.

What remains to be shown is that the remaining I/Os only contribute lower order terms or replace I/Os done on the canonical path. When an element travels through group G_{R-1} then $2/B$ I/Os must be charged for writing it to group buffer $R-1$ and later reading it when refilling the deletion buffer. However, the $2/B$ I/Os saved because the element is not moved to group G_R can pay for this charge. When an element travels through group buffer $i \leq R-2$, the additional $c \geq 2/B$ I/Os saved compared to the canonical path can also pay for the cost of swapping loser tree data for group G_i. The latter costs $2k(B+2)/B$ I/Os which can be divided among at least $m - m' \geq k(B+2)$ elements removed in one batch.

When group buffer $i \geq 2$ becomes invalid so that it must be merged with other group buffers and put back into group G_1, this causes a direct cost of $\mathcal{O}(m/B)$ I/Os and we must charge a cost of $\mathcal{O}(im/B)$ I/Os because these elements are thrown back $\mathcal{O}(i)$ steps on their path to the deletion buffer. Although an element may move through all the R groups we do not need to charge $\mathcal{O}(Rm/B)$ I/Os for small i since this only means that the shortcut originally taken by this element compared to the canonical path is missed. The remaining overhead can be charged to the $m(k-1)k^{j-2}$ insertions which have filled group G_{i-1}. Summing over all groups, each insertions gets an additional charge of $\sum_{i=2}^{R} \mathcal{O}(im/B)/(m(k-1)k^{j-2}) = \mathcal{O}(1/k)$. Similarly, invalidations of group buffer 1 give a charge $\mathcal{O}(1/k)$ per insertion.

We need $\mathcal{O}(\log k)$ I/Os for inserting a new sequence into the loser tree data structure. When done for tree 1, this can be amortized over m insertions. For tree $i > 1$ it can be amortized over $m(k^{i-1}-1)$ elements by Lemma 1. For an

[2] If we accept $\mathcal{O}(1/B)$ more I/Os per operation it would suffice to swap between the insertion buffer plus a constant number of buffer blocks and one loser tree with k sequence buffers in internal memory.

element moving on the canonical path, we get an overall charge of $\mathcal{O}(\log k/m) + \sum_{i=2}^{R} m(k^{i-1}-1)\log k = \mathcal{O}((\log k)/m)$ per insertion.

Overall we get a charge of $2R/B + \mathcal{O}(1/k + \log k/m)$. per insertion. ∎

We now estimate the number of key comparisons performed. We believe this is a good measure for the internal work since in efficient implementations of priority queues for the comparison model, this number is close to the number of unpredictable branch instructions (whereas loop control branches are usually well predictable by the hardware or the compiler) and the number of key comparisons is also proportional to the number of memory accesses. These two types of operations often have the largest impact on the execution time since they are the most severe limit to instruction parallelism in a super-scalar processor. In order to avoid notational overhead by rounding, we also assume that k and m are powers of two and that I is divisible by mk^{R-1}. A more general bound would only be larger by a small additive term.

Theorem 2. *With the assumptions from Theorem 1 at most* $I(\log I + \lceil \log R \rceil + \log m + 4 + m'/m + \mathcal{O}((\log k)/k))$ *key comparisons are needed. For average case inputs "*$\log m$*" can be replaced by* $\mathcal{O}(1)$.

Proof. Insertion into the insertion buffer takes $\log m$ comparisons at worst and $\mathcal{O}(1)$ comparisons on the average. Every `deleteMin` operation requires a comparison of the minimum of the insertion buffer and the deletion buffer. The remaining comparisons are charged to insertions in an analogous way to the proof of Theorem 1. Sorting the insertion buffer (e.g. using merge sort) takes $m \log m$ comparisons and merging the result with the deletion buffer and group buffer 1 takes $2m + m'$ comparisons. Inserting the sequence into a loser tree takes $\mathcal{O}(\log k)$ comparisons. Emptying groups takes $(R-1)\log k + \mathcal{O}(R/k)$ comparisons per element. Elements removed from the insertion buffer take up to $2\log m$ comparisons. But those need not be counted since we save all further comparisons on them. Similarly, refills of group buffers other than R have already been accounted for by our conservative estimate on group emptying cost. Group G_R only has degree $I/(mk^{R-1})$ so $\lceil \log I - (R-1)\log k - \log m \rceil$ comparisons per element suffice. Using similar arguments as in the proof of Theorem 1 it can be shown that inserting sequences into the loser trees leads to a charge of $\mathcal{O}((\log k)/m)$ comparisons per insertion and invalidating group buffers costs $\mathcal{O}((\log k)/k)$ comparisons per insertion. Summing all the charges made yields the bound to be proven. ∎

For external memory one would choose $m = \Theta(M)$ and $k = \Theta(M/B)$. In another paper we show that k should be a factor $\mathcal{O}(B^{1/a}/\delta)$ smaller on a-way associative caches in order to limit the number of cache faults to $(1+\delta)$ times the number of I/Os performed by the external memory algorithm. This requirement together with the small size of many first level caches and TLBs[3] explains why

[3] *T*ranslation *L*ook-aside *B*uffers store the physical position of the most recently used virtual memory pages.

we may have to live with a quite small k. This observation is the main reason why we did not pursue the simple variant of the array heap described in [7] which needs only a single merge group for all sequences. This merge group would have to be about a factor R larger however.

4 Refinements

Memory Management: A sequence heap can be implemented in a memory efficient way by representing sequences in the groups as singly linked lists of memory pages. Whenever a page runs empty, it is pushed on a stack of free pages. When a new page needs to be allocated, it is popped from the stack. If necessary, the stack can be maintained externally except for a single buffer block. Using pages of size p, the external sequences of a sequence heap with R groups and N elements occupy at most $N + kpR$ memory cells. Together with the measures described above for keeping the number of groups small, this becomes $N + kp \log_k N/m$. A page size of m is particularly easy to implement since this is also the size of the group buffers and the insertion buffer. As long as $N = \omega(km)$ this already guarantees asymptotically optimal memory efficiency, i.e., a memory requirement of $N(1 + o(1))$.

Many Operations on Small Queues: Let N_i denote the queue size before the i-th operation is executed. In the earlier algorithms [3, 7, 8] the number of I/Os is bounded by $\mathcal{O}(\sum_{i \le I} \log_k N_i/m)$. For certain classes of inputs, $\sum_{i \le I} \log_k N_i/m$ can be considerably less than $I \log_k I/m$. However, we believe that for most applications which require large queues at all, the difference will not be large enough to warrant significant constant factor overheads or algorithmic complications. We have therefore chosen to give a detailed analysis of the basic algorithm first and to outline an adaption yielding the refined asymptotic bound here: Similar to [7], when a new sequence is to be inserted into group G_i and there is no free slot, we first look for two sequences in G_i whose sizes sum to less than mk^{i-1} elements. If found, these sequences are merged, yielding a free slot. The merging cost can be charged to the `deleteMin`s which caused the sequences to get so small. Now G_i is only emptied when it contains at least $mk^i/2$ elements and the I/Os involved can be charged to elements which have been inserted when G_i had at least size $mk^{i-1}/4$. Similarly, we can "tidy up" a shrinking queue: When there are R groups and the total size of the queue falls below $mk^{R-1}/4$, empty group G_R and insert the resulting sequence into group G_{R-1} (if there is no free slot in group G_{R-1} merge any two of its sequences first).

Registers and Instruction Cache: In all realistic cases we have $R \le 4$ groups. Therefore, instruction cache and register file are likely to be large enough to efficiently support a fast R-way merge routine for refilling the deletion buffer which keeps the current keys of each stream in registers.

Second Level Cache: So far, our analysis assumes only a single cache level. Still, if we assume this level to be the first level cache, the second level cache may have some influence. First, note that the group buffers and the loser trees with their

group buffers are likely to fit in second level cache. The second level cache may also be large enough to accommodate all of group G_1 reducing the costs for $2/B$ I/Os per insert. We get a more interesting use for the second level cache if we assume its bandwidth to be sufficiently high to be no bottleneck and then look at inputs where deletions from the insertion buffer are rare (e.g. sorting). Then we can choose $m = \mathcal{O}(M_2)$ if M_2 is the size of the second level cache. Insertions have high locality if the $\log m$ cache lines currently accessed by them fit into first level cache and no operations on deletion buffers and group buffers use random access.

High Bandwidth Disks: When the sequence heap data structure is viewed as a classical external memory algorithm we would simply use the main memory size for M. But our measurements in Section 5 indicate that large binary heaps as an insertion buffer may be too slow to match the bandwidth of fast parallel disk subsystems. In this case, it is better to modify a sequence heap ooptimized for cache and main memory by using specialized external memory implementations for the larger groups. This may involve buffering of disk blocks, explicit asynchronous I/O calls and perhaps prefetching code and randomization for supporting parallel disks [2]. Also, the number of I/Os may be reduced by using a larger k inside these external groups. If this degrades the performance of the loser tree data structure too much, we can insert another heap level, i.e., split the high degree group into several low degree groups connected together over sufficiently large level-2 group buffers and another merge data structure.

Deletions of non-minimal elements can be performed by maintaining a separate sequence heap of deleted elements. When on a `deleteMin`, the smallest element of the main queue and the delete-queue coincide, both are discarded. Hereby, insertions and deletions cost only one comparison more than before, if we charge a delete for the costs of one insertion and two `deleteMin`s (note that the latter are much cheaper than an insertion). Memory overhead can be kept in bounds by completely sorting both queues whenever the size of the queue of deleted elements exceeds some fraction of the size of the main queue. During this sorting operation, deleted keys are discarded. The resulting sorted sequence can be put into group G_R. All other sequences and and the deletion heap are empty then.

5 Implementation and Experiments

We have implemented sequence heaps as a portable C++ template class for arbitrary key-value-pairs. Currently, sequences are implemented as a single array. The performance of our sequence heap mainly stems on an efficient implementation of the k-way merge using loser trees, special routines for 2-way, 3-way and 4-way merge and binary heaps for the insertion buffer. The most important optimizations turned out to be (roughly in this order): Making live for the compiler easy; use of *sentinels*, i.e., dummy elements at the ends of sequences and heaps which save special case tests; loop unrolling.

5.1 Choosing Competitors

When an author of a new code wants to demonstrate its usefulness experimentally, great care must be taken to choose a competing code which uses one of the best known algorithms and is at least equally well tuned. We have chosen implicit binary heaps and aligned 4-ary heaps. In a recent study [15], these two algorithms outperform the pointer based data structures splay tree and skew heap by more than a factor two although the latter two performed best in an older study [12]. Not least because we need the same code for the insertion buffer, binary heaps were coded perhaps even more carefully than the remaining components – binary heaps are the only part of the code for which we took care that the assembler code contains no unnecessary memory accesses, redundant computations and a reasonable instruction schedule. We also use the *bottom up heuristics* for `deleteMin`: Elements are first lifted up on a min-path from the root to a leaf, the leftmost element is then put into the freed leaf and is finally bubbled up. Note that binary heaps with this heuristics perform only $\log N + \mathcal{O}(1)$ comparisons for an insertions plus a `deleteMin` on the average which is close to the lower bound. So in flat memory it should be hard to find a comparison based algorithm which performs significantly better for average case inputs. For small queues our binary heaps are about a factor two faster than a more straightforward non-recursive adaption of the textbook formulation used by Cormen, Leiserson and Rivest [5].

Aligned 4-ary heaps have been developed at the end using the same basic approach as for binary heaps, in particular, the bottom up heuristics is also used. The main difference is that the data gets aligned to cache lines and that more complex index computations are needed.

All source codes are available electronically under `http://www.mpi-sb.mpg.de/~sanders/programs/`.

5.2 Basic Experiments

Although the programs were developed and tuned on SPARC processors, sequence heaps show similar behavior on all recent architectures that were available for measurements. We have run the same code on a SPARC, MIPS, Alpha and Intel processor. It even turned out that a single parameter setting – $m' = 32$, $m = 256$ and $k = 128$ works well for all these machines.[4] Figures 2, 3, 4 and 5 respectively show the results.

All measurements use random 32 bit integer keys and 32 bit values. For a maximal heap size of N, the operation sequence (`insert deleteMin insert`)^N (`deleteMin insert deleteMin`)^N is executed. We normalize the amortized execution time per `insert`-`deleteMin`-pair – $T/(6N)$ – by dividing by $\log N$. Since all algorithms have an "flat memory" execution time of $c \log N + \mathcal{O}(1)$ for some constant c, we would expect that the curves have a hyperbolic form and converge

[4] By tuning k and m, performance improvements around 10 % are possible, e.g., for the Ultra and the PentiumII, $k = 64$ are better.

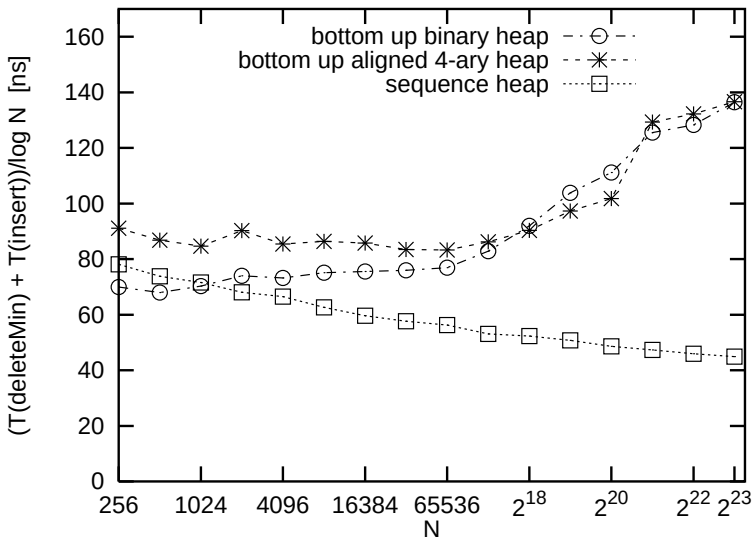

Fig. 2. Performance on a Sun Ultra-10 desktop workstation with 300 MHz UltraSparcIIi processor (1st-level cache: $M = 16$KByte, $B = 16$Byte; 2nd-level cache: $M = 512$KByte, $B = 32$Byte) using Sun Workshop C++ 4.2 with options `-fast -O4`.

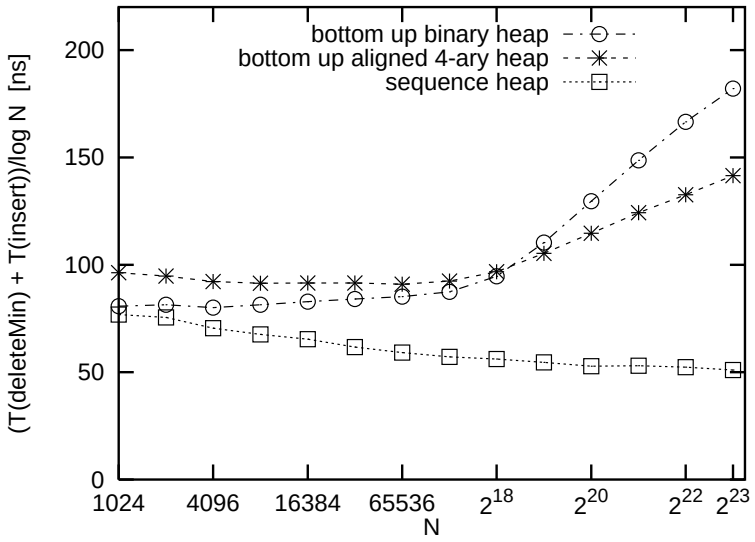

Fig. 3. Performance on a 180 MHz MIPS R10000 processor. Compiler: `CC -r10000 -n32 -mips4 -O3`.

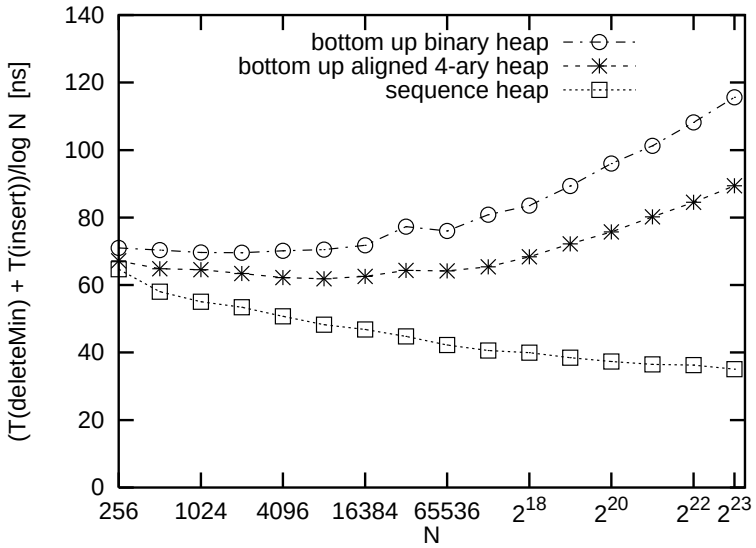

Fig. 4. Performance on a 533 MHz DEC-Alpha-21164 processor. Compiler: `g++ -O6`.

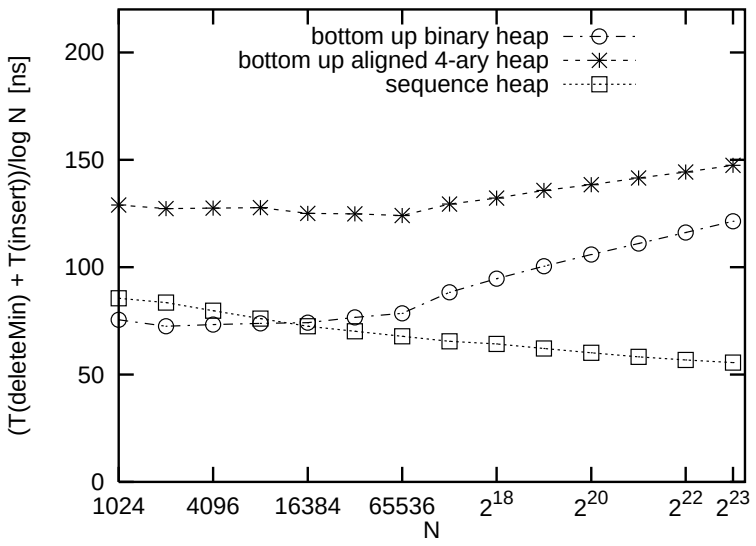

Fig. 5. Performance on a 300 MHz Intel Pentium II processor. Compiler: `g++ -O6`.

to a constant for large N. The values shown are averages over at least 10 trials. (More for small inputs to avoid problems due to limited clock resolution.) In order to minimize the impact of other processes and virtual memory management, a warm-up run is made before each measurement and the programs are run on (almost) unloaded machines.

Sequence heaps show the behavior one would expect for flat memory – cache faults are so rare that they do not influence the execution time very much. In Section 5.4, we will see that the decrease in the "time per comparison" is not quite so strong for other inputs.

On all machines, binary heaps are equally fast or slightly faster than sequence heaps for small inputs. While the heap still fits into second level cache, the performance remains rather stable. For even larger queues, the performance degradation accelerates. Why is the "time per comparison" growing about linearly in $\log n$? This is easy to explain. Whenever the queue size doubles, there is another layer of the heap which does not fit into cache, contributing a constant number of cache faults per `deleteMin`. For $N = 2^{23}$, sequence heaps are between 2.1 and 3.8 times faster than binary heaps.

We consider this difference to be large enough to be of considerable practical interest. Furthermore, the careful implementation of the algorithms makes it unlikely that such a performance difference can be reversed by tuning or use of a different compiler.[5] (Both binary heaps and sequence heaps could be slightly improved by replacing index arithmetics by arithmetics with address offsets. This would save a single register-to-register shift instruction per comparison and is likely to have little effect on super-scalar machines.) Furthermore, the satisfactory performance of binary heaps on small inputs shows that for large inputs, most of the time is spent on memory access overhead and coding details have little influence on this.

5.3 4-ary Heaps

The measurements in figures 2 through 5 largely agree with the most important observation of LaMarca and Ladner [15]: since the number of cache faults is about halved compared to binary heaps, 4-ary heaps have a more robust behavior for large queues. Still, sequence heaps are another factor between 2.5 and 2.9 faster for very large heaps since they reduce the number of cache faults even more. However, the relative performance of our binary heaps and 4-ary heaps seems to be a more complicated issue than in [15]. Although this is not the main concern of this paper we would like to offer an explanation:

Although the bottom up heuristics improves both binary heaps and 4-ary heaps, binary heaps profit much more. Now, binary heaps need less instead of more comparisons than 4-ary heaps. Concerning other instruction counts, 4-ary

[5] For example, in older studies, heaps and loser trees may have looked bad compared to pointer based data structures if the compiler generates integer division operations for halving an index or integer multiplications for array indexing.

heaps only save on memory write instructions while they need more complicated index computations.

Apparently, on the Alpha which has the highest clock speed of the machines considered, the saved write instructions shorten the critical path while the index computations can be done in parallel to slow memory accesses (spill code).

On the other machines, the balance turns into the other direction. In particular, the Intel architecture lacks the necessary number of registers so that the compiler has to generate a large number of additional memory accesses. Even for very large queues, this handicap is never made up for.

The most confusing effect is the jump in the execution time of 4-ary heaps on the SPARC for $N > 2^{20}$. Nothing like this is observed on the other machines and this effect is hard to explain by cache effects alone since the input size is already well beyond the size of the second level cache. We suspect some problems with virtual address translation which also haunted the binary heaps in an earlier version.

5.4 Long Operation Sequences

Our worst case analysis predicts a certain performance degradation if the number of insertions I is much larger than the size of the heap N. However, in Fig. 6 it can be seen that the contrary can be true for random keys.

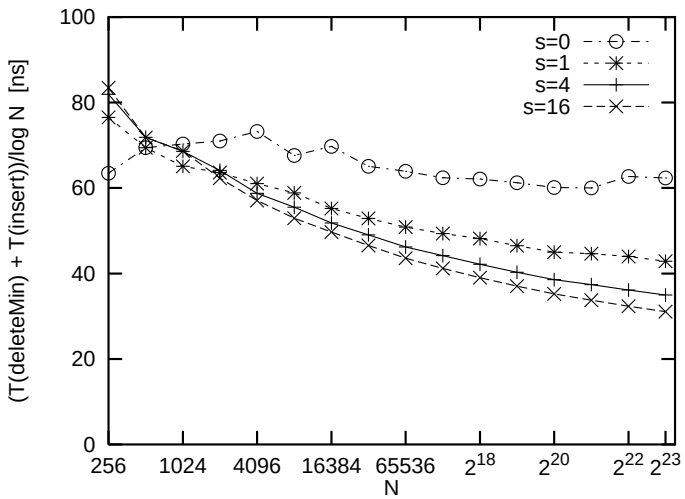

Fig. 6. Performance of sequence heaps using the same setup as in Fig. 2 but using different operation sequences: $(\texttt{insert}\ (\texttt{deleteMin}\ \texttt{insert})^s)^N$ $(\texttt{deleteMin}\ (\texttt{insert}\ \texttt{deleteMin})^s)^N$ for $s \in \{0, 1, 4, 16\}$. For $s = 0$ we essentially get heap-sort with some overhead for maintaining useless group and deletion buffers. In Fig. 2 we used $s = 1$.

For a family of instances with $I = 33N$ where the heap grows and shrinks very slowly, we are almost two times faster than for $I = N$. The reason is that new elements tend to be smaller than most old elements (the smallest of the old elements have long been removed before). Therefore, many elements never make it into group G_1 let alone the groups for larger sequences. Since most work is performed while emptying groups, this work is saved. A similar locality effect has been observed and analyzed for the Fishspear data structure [9]. Binary heaps or 4-ary heaps do not have this property. (They even seem to get slightly slower.) For $s = 0$ this locality effect cannot work. So that these instances should come close to the worst case.

6 Discussion

Sequence heaps may currently be the fastest available data structure for large comparison based priority queues both in cached and external memory. This is particularly true, if the queue elements are small and if we do not need deletion of arbitrary elements or decreasing keys. Our implementation approach, in particular k-way merging with loser trees can also be useful to speed up sorting algorithms in cached memory.

In the other cases, sequence heaps still look promising but we need experiments encompassing a wider range of algorithms and usage patterns to decide which algorithm is best. For example, for monotonic queues with integer keys, radix heaps look promising. Either in a simplified, average case efficient form known as calendar queues [4] or by adapting external memory radix heaps [6] to cached memory in order to reduce cache faults.

We have outlined how the algorithm can be adapted to multiple levels of memory and parallel disks. On a shared memory multiprocessor, it should also be possible to achieve some moderate speedup by parallelization (e.g. one processor for the insertion and deletion buffer and one for each group when refilling group buffers; all processors collectively work on emptying groups).

Acknowledgements

I would like to thank Gerth Brodal, Andreas Crauser, Jyrki Katajainen and Ulrich Meyer for valuable suggestions. Ulrich Rüde from the University of Augsburg provided access to an Alpha processor.

References

1. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *4th Workshop on Algorithms and Data Structures*, number 955 in LNCS, pages 334–345. Springer, 1995.
2. R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

3. Gerth Stølting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In *6th Scandinavian Workshop on Algorithm Theory*, number 1432 in LNCS, page 107ff. Springer Verlag, Berlin, 1998.
4. R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
6. A. Crauser and K. Mehlhorn et. al. On the performance of LEDA-SM. Technical Report MPI-I-98-1-028, Max Planck Institute for Computer Science, 1998.
7. A. Crauser, P. Ferragina, and U. Meyer. Efficient priority queues in external memory. working paper, October 1997.
8. R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. External heaps combined with effective buffering. In *4th Australasian Theory Symposium*, volume 19-2 of *Australian Computer Science Communications*, pages 72–78. Springer, 1997.
9. M. J. Fischer and M. S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.
10. J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 1996.
11. Intel Corporation, P.O. Box 5937, Denver, CO, 80217-9808, `http://www.intel.com`. *Intel Archtecture Software Developer's Manual. Volume I: Basic Architecture*, 1997. Ordering Number 243190.
12. D. Jones. An empirical comparison of priority-queue and event set implementations. *Communications of the ACM*, 29(4):300–311, 1986.
13. J. Keller. The 21264: A superscalar alpha processor with out-of-order execution. In *Microprocessor Forum*, October 1996.
14. D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 1973.
15. A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1(4), 1996.
16. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *8th ACM-SIAM Symposium on Discrete Algorithm*, pages 370–379, 1997.
17. MIPS Technologies, Inc. *R10000 Microprocessor User's Manual*, 2.0 edition, 1998. `http://www.mips.com`.
18. J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.
19. Sun Microsystems. *UltraSPARC-IIi User's Manual*, 1997.
20. D. E. Vengroff. *TPIE User Manual and Reference*, 1995. `http://www.cs.duke.edu/~dev/tpie_home_page.html`.
21. J. S. Vitter. External memory algorithms. In *6th European Symposium on Algorithms*, number 1461 in LNCS, pages 1–25. Springer, 1998.
22. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.
23. L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Transactions on Software Engineering*, 15(7):9–925, July 1989.