Alcom-FT Technical Report Series
ALCOMFT-TR-01-83

Fast Priority Queues for Cached Memory

Peter Sanders
Max-Planck-Institut for Computer Science,

The cache hierarchy prevalent in todays high performance processors has to be taken into account in order to design algorithms that perform well in practice. This paper advocates the adaption of external memory algorithms to this purpose. This idea and the practical issues involved are exemplified by engineering a fast priority queue suited to external memory and cached memory that 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.

Categories and Subject Descriptors: C.1.1 [**Computer Systems Organization**]: Single Data Stream Architectures; C.4 [**Computer Systems Organization**]: Performance of Systems; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations; E.5 [**Data**]: Files; F.2.2 [**Theory**]: Nonnumerical Algorithms and Problems; F.1.2 [**Theory**]: Modes of Computation

General Terms: Data structure, Cache, Implementation

Additional Key Words and Phrases: Priority queue, External memory, Secondary storage, Cache efficiency, Multi way merging, Heap

1. INTRODUCTION

The mainstream model of computation used by algorithm designers in the last half century [Neumann 1945] 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 [Hennessy and Patterson 1996; Intel Corporation 1997; Keller 1996; MIPS Technologies, Inc. 1998; Sun Microsystems 1997]. In particular, we usually distinguish at least four levels of memory hierarchy: A file of multiported *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

Address: Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany. E-mail: `sanders@mpi-sb.mpg.de`
WWW: `http://www.mpi-sb.mpg.de/~sanders`
Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

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 inner loops of 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 [Vitter and Shriver 1994]. This model assumes an internal memory of size M that can access the external memory by transferring blocks of size B. The word pairs "cache line" and "memory block", "cache" and "internal memory", "main memory" and "external memory", and "I/O" and "cache fault" are used 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. For the kind of algorithms considered here, this mainly has the effect of reducing the usable cache size by a factor $B^{1/a}$ where B is the memory block size and a is the *associativity* of the cache [Sanders 1999].[1]

Henceforth, the term *cached memory* is used in order to make clear that we have a different model.

Despite 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, care must be taken 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 should prefer algorithms that use the cache as economically as possible. Finally, the remaining levels of the memory hierarchy are discussed informally in order to keep the analysis focussed on the most important aspects.

Section 2 presents the basic algorithm for the *sequence heaps* data structure for priority queues[2]. 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 `deleteMins` 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

[1] In an *a-way set associative cache* every memory block is mapped to a fixed *cache set* and every cache set can hold at most a blocks.

[2] A priority queue is a data structure for representing a totally ordered set that supports insertion of elements and deletion of the minimal element.

comparisons. Similar bounds hold for cached memory with a-way associative caches if k is reduced by $\mathcal{O}(B^{1/a})$ [Sanders 1999]. Section 4 considers refinements that 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. An appendix gives further details on the implementation. Its goal is to make the results easier to reproduce in other contexts and to argue that all considered codes are comparably efficiently implemented.

Related Work

External memory algorithms are a well established branch of algorithmics (e.g. [Vitter 1998; Vengroff 1995]). The external memory heaps of Teuhola and Wegner [Wegner and Teuhola 1989] and the fishspear data structure [Fischer and Paterson 1994] need a factor $\Theta(B)$ fewer I/Os than traditional priority queues such as binary heaps. Buffer search trees [Arge 1995] were the first external memory priority queues 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. [Brodal and Katajainen 1998; Brengel et al. 1999; Fadel et al. 1997] are more directly geared to priority queues and achieve the same asymptotic bounds, one even per operation and not in an amortized sense [Brodal and Katajainen 1998]. A sequence heap is very similar. In particular, it can be considered a simplification and reengineering of the "improved array-heap" [Brengel et al. 1999]. However, sequence heaps are more I/O-efficient by a factor of about three (or more) than [Arge 1995; Brodal and Katajainen 1998; Brengel et al. 1999; Fadel et al. 1997] and need about a factor of two less memory than [Arge 1995; Brengel et al. 1999; Fadel et al. 1997].

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 [Knuth 1973] and cached memory [LaMarca and Ladner 1997]. 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 that 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.

A separate binary heap with capacity m allows an arbitrary mix of insertions and deletions by holding 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 the sequences for the k-way merge.

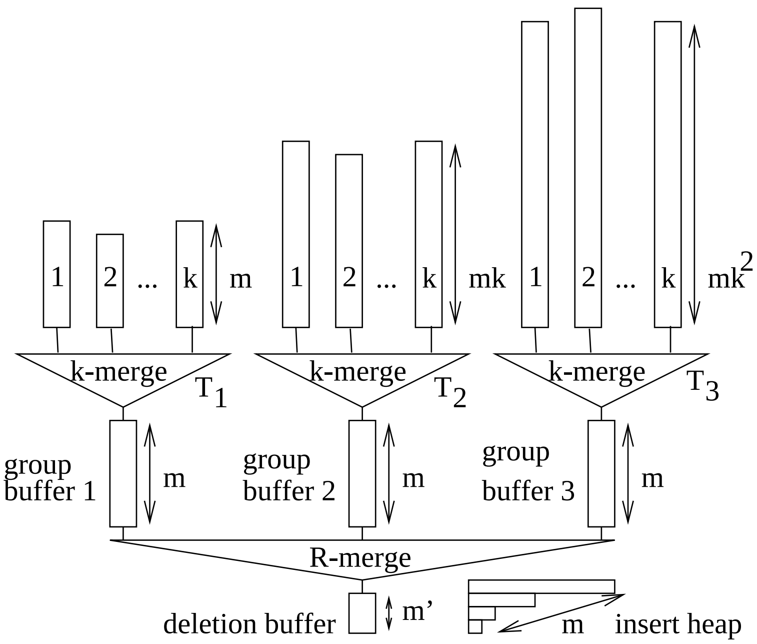

Fig. 1. Overview of the data structure for sequence heaps for $R = 3$ merge groups.

Up to this point, sequence heaps and several earlier data structures [Brodal and Katajainen 1998; Brengel et al. 1999; Fadel et al. 1997] 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 make room by merging all the k sequences producing a larger sequence of size up to km [Brodal and Katajainen 1998; Brengel et al. 1999].

Now the question arises how to handle the larger sequences. Sequence heaps adopt the approach used for *improved array-heaps* [Brengel et al. 1999] 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 understand 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 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 [Brengel et al. 1999] where the invariant is maintained by merging the new sequence and the group buffer, which almost doubles the total number of I/Os.

For cached memory, where the speed of internal computation matters, it is also crucial how to implement the operation of k-way merging. The "loser tree" variant of the selection tree data structure described by Knuth [Knuth 1973, Section 5.4.1] is particularly well suited: 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 [Brengel et al. 1999; Fadel et al. 1997] 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

The analysis first quantifies 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 `deleteMins`. The analysis continues with the number of key comparisons as a measure of internal work and then discusses how k, m, and m' should be chosen for external memory and cached memory respectively. Adaptations 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

$$\sum_{j=1}^{R} m(k-1)k^{j-1} = m(k^i - 1)$$

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.*

The analysis counts 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 and G_{R-1} (assuming that $k(B+2)$ units of memory suffice to store the blocks of the k sequences that are currently accessed and the loser tree information itself); a corresponding amount of space for one more loser tree shared by the remaining $R-2$ groups and data for merging the R group buffers.[3]

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, $1/B$ I/Os are charged to the insertion of this element. Then it is involved in emptying groups until it arrives in group G_R. For each emptying operation, the element is involved into one batched read and one batched write, i.e., it is charged with $2(R-1)/B$ I/Os for tree emptying operations. Eventually, the element is read into group buffer R yielding a charge of $1/B$ I/Os for. 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

[3] 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.

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 a cost of $\mathcal{O}(im/B)$ I/Os must be charged 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 that 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 of $\mathcal{O}(1/k)$ per insertion.

Inserting a new sequence into the loser tree takes data structure takes $\mathcal{O}(\log k)$ I/Os. 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 element moving on the canonical path, we get an overall charge of

$$\mathcal{O}\left(\frac{\log k}{m}\right) + \sum_{i=2}^{R} \frac{\log k}{m(k^{i-1}-1)} = \mathcal{O}\left(\frac{\log k}{m}\right)$$

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

An estimate the number of key comparisons performed comes next. 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, 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 an m element 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 no further comparisons are needed for them. Similarly, refills of group buffers other than R have already been accounted for by the 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 $k = \Theta(M/B)$. For cached memory with an a-way associative cache, k should be a factor $\Theta(B^{1/a}/\delta)$ smaller in order to limit the number of cache faults to $(1+\delta)$ times the number of I/Os performed by the external memory algorithm [Sanders 1999]. This requirement together with the small size of many first level caches and TLBs[4] explains why we may have to live with a quite small k. This observation is the main reason not to pursue the simple variant of the array heap described in [Brengel et al. 1999] that 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 below 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 other algorithms [Brodal and Katajainen 1998; Brengel et al. 1999; Fadel et al. 1997] the number of I/Os is bounded by $\mathcal{O}(\sum_{i\le I}\log_k N_i/m)$. For some classes of inputs, $\sum_{i\le I}\log_k N_i/m$ can be considerably less than $I\log_k I/m$. However, for most applications that require large queues at all, the difference should not be large enough to warrant significant constant factor overheads or algorithmic complications. Therefore this paper gave a detailed analysis of the basic algorithm first and outlines an adaptation yielding the refined asymptotic bound here: Similar to [Brengel et al. 1999], when a new sequence is to be inserted into group G_i and

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

there is no free slot, 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 `deleteMins` that 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 that have been inserted when G_i had at least size $mk^{i-1}/4$. Similarly, a shrinking queue can be "tidied up": 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 $R \leq 4$ 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. Refer to Appendix A.4 for more details.

Second Level Cache:. So far, the 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 k sequence buffer blocks each 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. Furthermore, no operations on deletion buffers and group buffers use random access.

High Bandwidth Disks:. If the sequence heap data structure is viewed as a classical external memory algorithm we would simply use the main memory size for M. In this case large binary heaps would be used as insertion buffers. But the measurements in Section 5 indicate that large binary heaps might be too slow to match the bandwidth of fast parallel disk subsystems. In this case, it is better to modify a sequence heap optimized 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 [Barve et al. 1997]. 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 with known key value 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 `deleteMins` cost only one comparison more than before, if we charge a delete for the costs of one insertion and two `deleteMins` (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 the deletion heap are empty then.

5. IMPLEMENTATION AND EXPERIMENTS

Sequence heaps were implemented as a portable C++ template class for arbitrary key-value-pairs. Currently, sequences are implemented as a single array. The internal performance mainly depends 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. The appendix discusses these issues in detail.

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 that uses one of the best known algorithms and is at least equally well tuned. Implicit binary heaps and aligned 4-ary heaps were chosen. In a recent study [LaMarca and Ladner 1996], these two array based algorithms outperform the pointer based data structures splay tree and skew heap by more than a factor of two. The situation is complicated by fact that the above pointer based algorithms performed best in an older study [Jones 1986]. All four algorithms execute a similar number of instructions but pointer based algorithms need up to three times more memory references. Possibly, the growing gap between memory speed and processor speed has tipped the balance in favor of array based algorithms.

Not least because the same code is needed 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 care was taken that the assembler code contains no unnecessary memory accesses, redundant computations, and a reasonable instruction schedule. The code also uses the *bottom up heuristic* [Wegener 1993] 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 heuristic perform only $\log N + \mathcal{O}(1)$ comparisons for an insertions plus a `deleteMin` on the average. This is close to the lower bound. So in flat memory it should be hard to find a comparison based algorithm that performs significantly better for average case inputs. For small queues the implementation of binary heaps is about a factor two faster than a more straightforward non-recursive adaptation of the textbook formulation used by Cormen, et al. [Cormen et al. 1990]. Appendix A.1 discusses the implementation of binary heaps in detail.

Aligned 4-ary heaps have been developed at the end using the same basic approach as for binary heaps, in particular, the bottom up heuristic is also used. The main difference is that the data gets aligned to cache lines and that more complex index computations are needed. Appendix A.2 gives more details.

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. The same code was run 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.[5] 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 $(\texttt{insert deleteMin insert})^N$ $(\texttt{deleteMin insert deleteMin})^N$ is executed. To normalize the amortized execution time per `insert`-`deleteMin`-pair, $T/(6N)$, it is divided 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 to a constant for large N. The values shown are averages over at least 10 repetitions.[6] The programs where run on an unloaded machine. Timing uses the operating system function for returning the CPU time consumed by the user process. To make sure that everything is in internal memory and warmup the caches, one initial execution was performed without timing it.

[5]By tuning k and m, performance improvements around 10 % are possible, e.g., for the Ultra and the PentiumII, $k = 64$ is better.
[6]More for small inputs to avoid problems due to limited clock resolution.

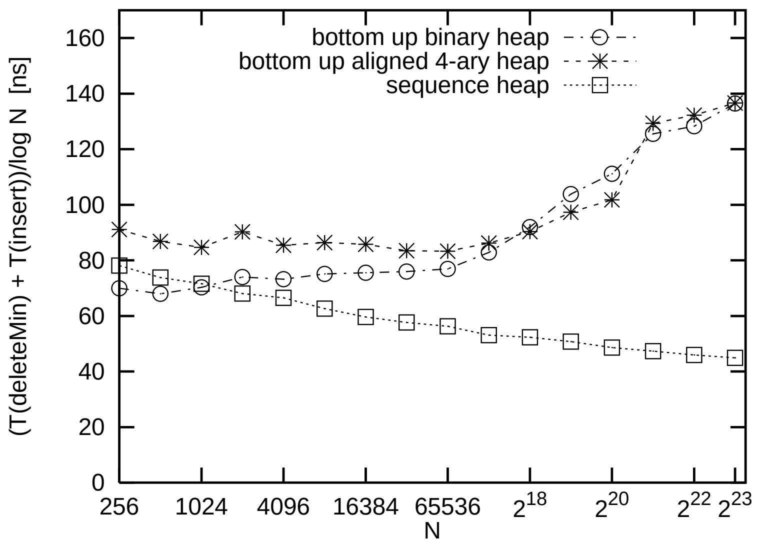

Fig. 2. Performance on a Sun Ultra-10 desktop workstation with 300 MHz Ultra-SparcIIi processor (1st-level cache: $M = 16$KByte, $B = 16$Byte; 2nd-level cache: $M = 512$KByte, $B = 32$Byte) using Sun Workshop C++ 4.2 with options `-fast -O4`.

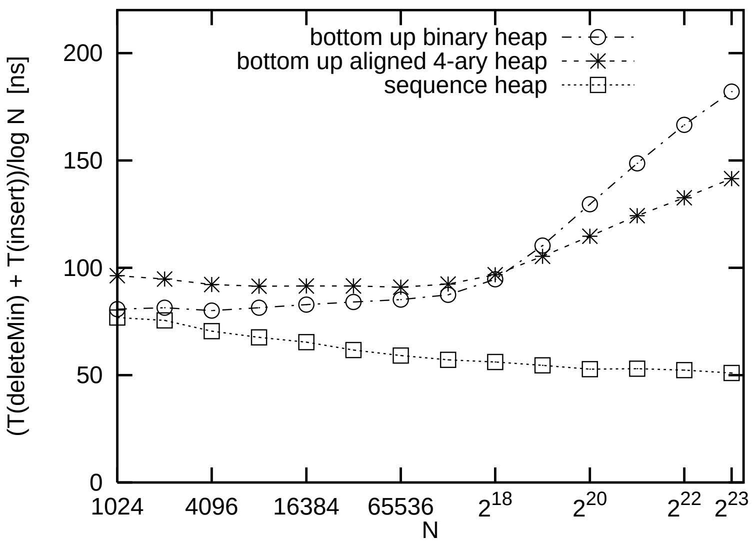

Fig. 3. Performance on a 180 MHz MIPS R10000 processor. Compiler: `CC -r10000 -n32 -mips4 -O3`.

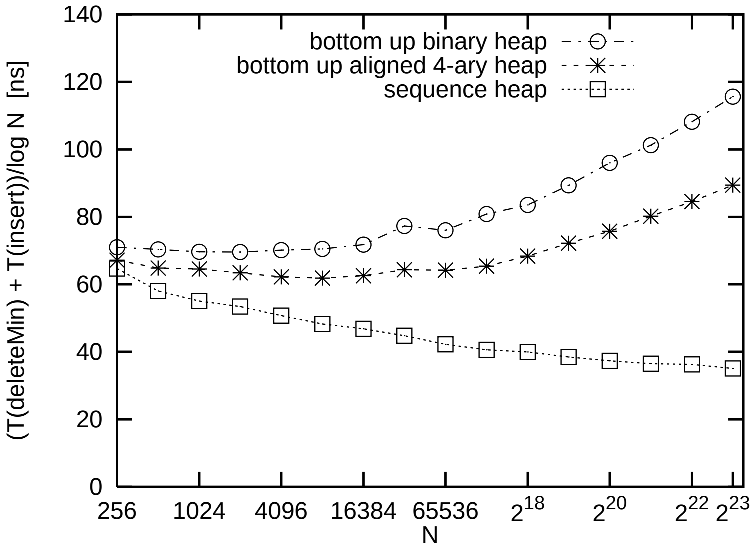

Fig. 4. Performance on a 533 MHz DEC-Alpha-21164 processor. Compiler: `g++ -O6`.

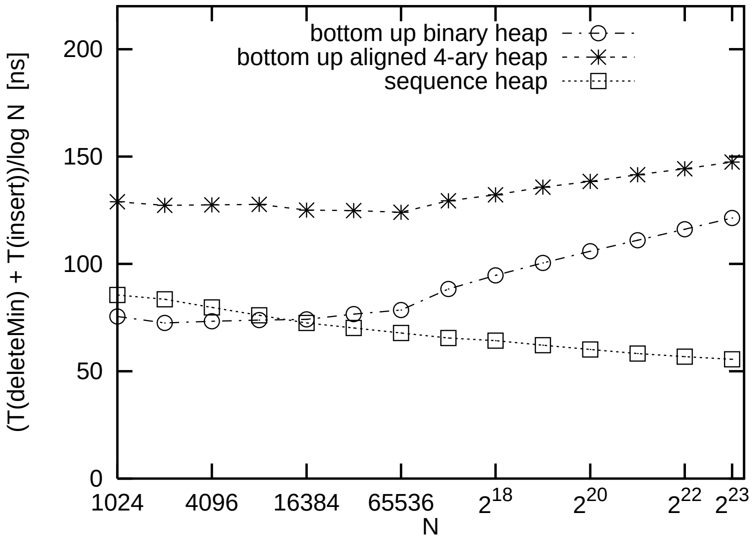

Fig. 5. Performance on a 300 MHz Intel Pentium II processor. Compiler: `g++ -O6`.

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 degenerates. This is easy to explain. Whenever the queue size doubles, there is another layer of the heap that 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.

This difference is 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.[7] 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 [LaMarca and Ladner 1996]: Since the number of cache faults is about halved compared to binary heaps, 4-ary heaps have a more

[7]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.

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 binary heaps and 4-ary heaps seems to be a more complicated issue than in [LaMarca and Ladner 1996]. Although this is not the main concern of this paper a possible explanation should be offered:

Although the bottom up heuristic 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 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.

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 (spill code). 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 but still below the main memory size. Possibly, there is a problem 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 independent keys.

For a family of instances with $I = 33N$ where the heap grows and shrinks very slowly, sequence heaps are about 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 [Fischer and Paterson 1994]. 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. To make clear that sequence heaps are nevertheless still much better than binary or 4-ary heaps, Figure 6 additionally contains their timing for $s = 0$.

Another way to generate inputs without locality effects is to use *monotone* inputs, i.e., keys that are larger than all keys of previously deleted elements. This can be achieved by adding a random offset to the most recently deleted element. This model has also been tried for reasons of comparability with earlier studies [Jones 1986; LaMarca and Ladner 1996]. The results are not shown here however, since they are so similar to the case $s = 0$ shown above.

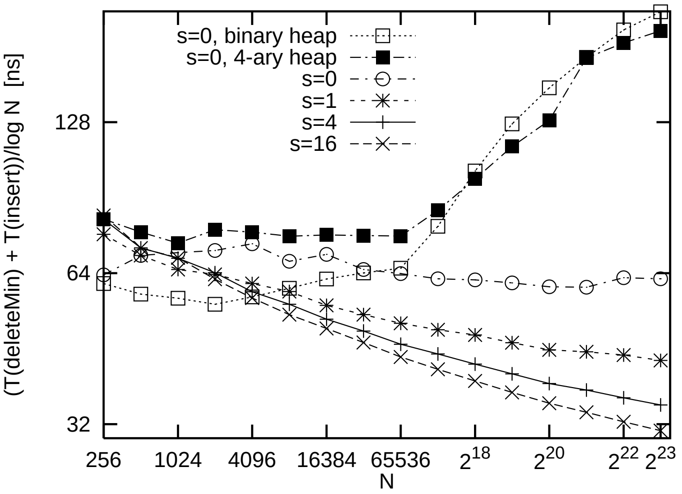

Fig. 6. Performance of sequence heaps using the same setup as in Fig. 2 but using different operation sequences: (`insert` (`deleteMin insert`)^s)^N (`deleteMin` (`insert deleteMin`)^s)^N for $s \in \{0, 1, 4, 16\}$. For $s = 0$ we essentially get heap-sort with some overhead for maintaining useless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

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 [Brown 1988] or by adapting external memory radix heaps [Brengel et al. 1999] to cached memory in order to reduce cache faults.

It has been 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).

ACKNOWLEDGMENTS

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.

APPENDIX

A. IMPLEMENTATION DETAILS

The experiments in Section 5 evaluate the different algorithms based on speedup margins between two and four. Although this is a quite useful for practical applications, such a margin is only of scientific interest if it is reproducible and understandable. It should be possible to understand how this margin can be achieved, and it must be possible to write similar codes even in other programming languages and using different machines and compilers. A doubtful researcher must have a chance to point her finger at some piece of code or of the experimental procedure and demonstrate that this particular detail causes the speedup margin thereby falsifying different claims made in the original publication. Therefore, the source code used for the experiments in this paper is made publicly available for academic use[8]. In addition, the following sections explain implementation details of the inner loops governing performance in the implemented codes. Perhaps they even succeed to demonstrate that all codes are hardly improvable on current technology. Although such high level of sophistication is no end in itself in an experimental study, it is nevertheless crucial for the validity of the experiments. Comparing several algorithms with different degree of optimization would certainly be questionable. But it is very difficult to argue that two codes are "equally suboptimal". One consequence of this observation is that ill-documented codes written by inexperienced programmers are of little experimental value if they do not show very clear margins. For example, the optimizations for binary heaps described below yield speedups compared to straight-forward implementations by more than a factor of two. If we would have compared the slow implementation with a tuned implementation of another algorithm, a speedup of two would have been meaningless with respect to the quality of the algorithms.

A.1 Binary Heaps

The average case complexity of insertion is constant so that deletion dominates the time required by binary heaps and by the insertion buffer of sequence heaps. Therefore only the implementation of `deleteMin` is considered here.

A.1.1 *Textbook-Style Implementation.* Before looking at the binary heap code used in the experiments, consider an implementation, which might be expected from a student looking at a textbook. The following code is a nonrecursive adaptation of the code proposed in the well known textbook of Cormen et al. [Cormen et al. 1990].

```
template <class Key, class Value>
inline void Heap2<Key, Value>::deleteMinBasic()
{ int i, l, r, smallest;

  data[1] = data[size];
  data[size].key = getSupremum();          /* infinity */
  size--;  i = 1;
```

[8] http://www.mpi-sb.mpg.de/~sanders/programs/

```
  for (;;) {
    l = (i << 1);  r = (i << 1) + 1;
    if ((l <= size) && data[l].key < data[i].key) {
      smallest = l;
    } else {
      smallest = i;
    }
    if ((r <= size) && data[r].key < data[smallest].key) {
      smallest = r;
    }
    if (smallest == i) break;
    Element temp = data[i];
    data[i] = data[smallest];
    data[smallest] = temp;
    i = smallest;
  }
}
```

For a queue of size N, the loop is traversed $\log N - \mathcal{O}(1)$ times on the average. Each iteration requires five conditional branches, two of which are key comparisons whose outcome is difficult to predict. Even at the highest level of compiler optimization, the SUN compiler generated 17 memory accesses in the inner loop if key and data fields consist of one memory word each.

A.1.2 *Optimized Implementation.* The optimized code starts as follows:

```
template <class Key, class Value, int capacity>
inline void BinaryHeap<Key, Value, capacity>::
deleteMin()
{ int hole = 1,  succ = 2,  sz = size;
```

First note, that the member variable `size` is copied into a local variable. This is done with every member variable that is used more than twice. Even using the highest level of compiler optimization, both the GNU `g++` and Sun's proprietary compiler were not able to appropriately put member variables into registers. (This measure alone reduces the number of memory references in the inner to 11.) After initialization, the element at the root of the heap is moved to a leaf such that the heap property is maintained. Note that there is only a single comparison for breaking the loop, since there are always $\log \texttt{sz}$ iterations. In addition, this comparison is easy to predict so that it does not stall the pipelines of modern processors [Hennessy and Patterson 1996].

No loop unrolling is done, since saving loop control overhead is only a small incentive on superscalar processors. Furthermore, there is no apparent opportunity for better instruction scheduling here, since the memory locations involved are highly data dependent.

```
  while (succ < sz) {
    Key key1 = data[succ].key;
    Key key2 = data[succ + 1].key;
    if (key1 > key2) { succ++;
```

```
      data[hole].key   = key2;
      data[hole].value = data[succ].value;
    } else {
      data[hole].key   = key1;
      data[hole].value = data[succ].value;
    }
    hole = succ;
    succ <<= 1;
  }
```

Note that there is only a single additional conditional branch in the body of the loop. In codes without the bottom up heuristic, there are at least two key comparisons with associated branches. On the average, code without the bottom up heuristic performs only a constant number less iterations.

In order to validate that good code can be generated for the binary heap algorithm, the assembler code was checked until none of the inner loops of binary heaps showed obvious shortcomings of the compiler (for the SUN and its native compiler). Consider the following annotated version of the assembler output produced by the compiler for the first inner loop of `deleteMin`:

```
.L900000807:        // label starting inner loop
ld [%g2+%g3],%o5    // Key key1 = data[succ].key;
ld [%g2+%o2],%o4    // Key key2 = data[succ + 1].key;
cmp %o4,%o5         // key1 <= key2 ?
ble,a .L900000808   // yes? branch after next instr.
sll %o1,3,%o5       // compute address offset for data[hole].key
// then-part of inner loop
  sll %o1,3,%g2     // compute address offset for data[hole].key
  add %o0,1,%o1     // succ++;
  st  %o5,[%g2+%o2]// data[hole].key   = key2;
  sll %o1,3,%o5     // address offset for data[succ].value
  ld [%o5+%o3],%o5 // register_o5 = data[succ].value;
  sll %o1,1,%o0     // succ <<= 1;
  st %o5,[%g2+%o3] // data[hole].value = register_o5;
  ba .L77000423     // branch around else part after next instr.
  cmp %o0,%g4       // succ < size? (delay slot)
.L900000808:        // else part of inner loop
  st %o4,[%o5+%o2] // data[hole].key   = key1;
  or %g0,%o0,%o1
  ld [%g2+%o3],%g2 // register_g2 = data[succ].value;
  sll %o0,1,%o0     // succ <<= 1;
  st %g2,[%o5+%o3] // data[hole].value = register_g2
  cmp %o0,%g4       // succ < size?

.L77000423:         // end of if () {} else {}
bl .L900000807      // next iteration if succ < size
sll %o0,3,%g2       // compute address offset for data[succ].key
```

In particular, note that only five memory accesses are performed for each iteration of this inner loop.

One possible optimization in the above loop concerns the index arithmetics. Using assembler or a somewhat awkward formulation in C, it would be possible to save two of the three left-shift (`sll`) instructions executed per loop iteration. Currently, these instructions are needed for converting an index into an address offset[9]. However, most of the time these instructions are not on the critical path for instruction scheduling, and on a superscalar machine, saving the shift instructions might not affect execution time at all. In any case, compared to the cost of the unavoidable, hard to predict branch their cost is negligible on most modern architectures.

What remains to be done is to overwrite the deleted element with the rightmost element of the heap and to move this element up in the tree to restore the heap property.

```
  Key bubble = data[sz].key;
  int pred = hole >> 1;
  while (data[pred].key > bubble) {
    data[hole] = data[pred];
    hole = pred;
    pred >>= 1;
  }

  // finally move data to hole
  data[hole].key = bubble;
  data[hole].value = data[sz].value;
```

Note that on the average, this loop only performs a constant number of iterations. Also, it causes no cache faults, since all the required cache lines were already moved into the cache when descending the tree.

At last, the now empty element at the right end of the tree is turned into a *sentinel element* by setting its key to infinity. Likewise, there is a sentinel element to the left of the root of the heap with key minus infinity. These measures make the fast and simple formulation of the inner loops possible that do not need to test for special cases at the border of the data structure. Compared to the formulation in Cormen et al. sentinels save two (easy to predict) branches for each loop iteration.

```
  data[size].key = getSupremum(); // mark as deleted
  size = sz - 1;
}
```

A remark on the bottom up heuristic is in order. It is a bit slower than the ordinary code in the worst case, and its advantage is not very large (less than 20%) compared to a careful formulation of the ordinary algorithm that has only one hard to predict branch in the inner loop: If the left and right successor are compared first, the comparison with the sifted element can only yield a 'smaller' once per deleteMin. This means that the relative performance of binary heaps and sequence heaps is not much affected by the presence or absence of the bottom up heuristic.

[9] Is is assumed here that the size of a heap element is a power of two. In a production implementation, this should perhaps be made sure by padding each element to the next power of two.

A.2 Aligned 4-ary Heaps

Our implementation follows the description of LaMarca and Ladner [LaMarca and Ladner 1996] as far as possible except that the bottom up heuristic is also used for 4-ary heaps. Here is the only loop traversed more than a constant number of times for an operation on the average:

```
while (succ < sz) {
  minKey = data[succ].key;
  delta = 0;

  otherKey = data[succ + 1].key;
  if (otherKey < minKey) { minKey = otherKey; delta = 1; }
  otherKey = data[succ + 2].key;
  if (otherKey < minKey) { minKey = otherKey; delta = 2; }
  otherKey = data[succ + 3].key;
  if (otherKey < minKey) { minKey = otherKey; delta = 3; }
```

The array data is aligned in such a way that the above three memory accesses concern a minimal number of cache lines.

```
  succ += delta;
  layerPos += delta;

  // move min successor up
  data[hole].key = minKey;
  data[hole].value = data[succ].value;

  // step to next layer
  hole = succ;
  succ = succ - layerPos + layerSize; // beginning of next layer
  layerPos <<= 2;
  succ += layerPos; // now correct value
  layerSize <<= 2;
}
```

The loop is executed a factor of two less frequently but there are a factor of three more hard to predict branches per loop iteration. In addtion, the index arithmetics is more complicated.[10]

A.3 Two-Way Merging

The function `merge` is needed when the insertion buffer is emptied into group one and for refilling the deletion buffer when $R = 2$. The function takes two sorted input sequences that are terminated by a sentinel element with infinite key. As in the case of binary heaps, these sentinels save special case treatments in the inner loop.

[10] Those could be somewhat simplified at the cost of leaving holes in the data structure. This could cause additional cache faults however.

```
// merge sz element from the two sentinel terminated input
// sequences *f0 and *f1 to "to"
// advance *f0 and *f1 accordingly.
// require: at least sz nonsentinel elements available in f0, f1
// require: to
template <class Key, class Value>
void merge(KNElement<Key, Value> **f0,
           KNElement<Key, Value> **f1,
           KNElement<Key, Value>  *to, int sz)
{
```

The parameters are double-indirect pointers, since they are also return values. Inside the function, one level of indirection is removed by keeping the current positions in a register:

```
  KNElement<Key, Value> *from0   = *f0;
  KNElement<Key, Value> *from1   = *f1;
  KNElement<Key, Value> *done    = to + sz;
  Key      key0    = from0->key;
  Key      key1    = from1->key;
```

As before, many other values are kept in registers for optimization purposes. In a specialized implementation for PC-processors some of these optimizations might turn out contraproductive, since not enough registers are available.

The main loop is very simple. Only one hard to predict plus one easy to predict branch per loop iteration is required.

```
  while (to < done) {
    if (key1 <= key0) {
      to->key   = key1;
      to->value = from1->value;
      from1++;
      key1 = from1->key;
    } else {
      to->key   = key0;
      to->value = from0->value;
      from0++;
      key0 = from0->key;
    }
    to++;
  }
```

Again, loop unrolling did not appear promising.

Finally, the positions of the source sequences are updated.

```
  *f0   = from0;
  *f1   = from1;
}
```

A.4 3-Way and 4-Way Merging

Since the number of merge groups R is typically bounded by four, it makes sense to supply specialized routines that can be used for refilling the deletion buffer when $R = 3$ and $R = 4$. The instruction caches of todays processors are large enough to implicitly store the relative order of the heads of the sequences in the program counter and to keep the keys of the elements in registers. Here is an example for the case of 3-way merging.

```
template <class Key, class Value>
void merge3(KNElement<Key, Value> **f0,
            KNElement<Key, Value> **f1,
            KNElement<Key, Value> **f2,
            KNElement<Key, Value>  *to, int sz)
{
  KNElement<Key, Value> *from0   = *f0;
  KNElement<Key, Value> *from1   = *f1;
  KNElement<Key, Value> *from2   = *f2;
  KNElement<Key, Value> *done    = to + sz;
```

To set up the merging process, the first keys of each sequence are copied into registers and their relative order is found out.

```
  Key      key0    = from0->key;
  Key      key1    = from1->key;
  Key      key2    = from2->key;
  if (key0 < key1) {
    if (key1 < key2)   { goto s012; }
    else {
      if (key2 < key0) { goto s201; }
      else             { goto s021; }
    }
  } else {
    if (key1 < key2) {
      if (key0 < key2) { goto s102; }
      else             { goto s120; }
    } else             { goto s210; }
  }
```

For each of the six possible relative orders `keya` $\leq$ `keyb` $\leq$ `keyc` an analogous piece of code must be provided. The most reasonable way to do that is a macro, since an inline function would not have direct access to all required local variables. The token pasting property of ANSI-C is useful here, since it makes it possible to build variable names and goto-labels from macro parameters.

```
#define Merge3Case(a,b,c)\
  s ## a ## b ## c :\
  if (to == done) goto finish;\
  to->key = key ## a;\
  to->value = from ## a -> value;\
```

```
  to++;\
  from ## a ++;\
  key ## a = from ## a -> key;\
  if (key ## a < key ## b) goto s ## a ## b ## c;\
  if (key ## a < key ## c) goto s ## b ## a ## c;\
  goto s ## b ## c ## a;

  // the order of the cases is chosen
  // in such a way that  four of the trailing gotos
  // can be eliminated by the optimizer
  Merge3Case(0, 1, 2);
  Merge3Case(1, 2, 0);
  Merge3Case(2, 0, 1);
  Merge3Case(1, 0, 2);
  Merge3Case(0, 2, 1);
  Merge3Case(2, 1, 0);

 finish:
  *f0   = from0;
  *f1   = from1;
  *f2   = from2;
}
```

A.5 k-Way Merging with Loser Trees

Loser trees are implemented as a C++ class that stores the loser tree itself and pointers to the sequences attached to its leaves.

```
// multi-merge for arbitrary K
template <class Key, class Value>
void KNLooserTree<Key, Value>::
multiMergeK(Element *to, int l)
{ Entry *currentPos;
  Key currentKey;
  int currentIndex; // leaf pointed to by current entry
  int kReg = k;
  Element *done = to + l;
```

The smallest element of all sequences is stored in the root of the tree. Therefore, finding the globally smallest element is easy:

```
  int       winnerIndex = entry[0].index;
  Key       winnerKey   = entry[0].key;
  Element *winnerPos;
  Key sup = dummy.key; // supremum
  while (to < done) {
    winnerPos = current[winnerIndex];

    // write result
    to->key   = winnerKey;
```

```
    to->value = winnerPos->value;

    // advance winner segment
    winnerPos++;
    current[winnerIndex] = winnerPos;
    winnerKey = winnerPos->key;

    // remove winner segment if empty now
    if (winnerKey == sup) {
      deallocateSegment(winnerIndex);
    }
```

After the smallest element has been consumed, the invariant of the data structure must be reestablished. This operation is the inner loop dominating the execution time of sequence heaps, since it is needed whenever merge groups are emptied or group buffers are refilled.

```
    // go up the entry-tree
    for (int i = (winnerIndex + kReg) >> 1;  i > 0;  i >>= 1) {
      currentPos = entry + i;
      currentKey = currentPos->key;
      if (currentKey < winnerKey) {
        currentIndex      = currentPos->index;
        currentPos->key   = winnerKey;
        currentPos->index = winnerIndex;
        winnerKey         = currentKey;
        winnerIndex       = currentIndex;
      }
    }
```

If one inspects the assembler code for this inner loop and of the inner loop of delete-min for bottom up binary heaps, it turns out that both contain about the same number of instructions and both are executed about the same number of times on the average. Loser trees have two advantage however. First, they do not need more time even in the worst case (So for worst case inputs we might find an even larger advantage for sequence heaps.) Second, the memory locations accessed are completely defined by the index of the leaf under considerations. Therefore, loop unrolling turned out to be profitable here. (At least on the Ultra-Sparc processor).

```
    to++;
  }
  entry[0].index = winnerIndex;
  entry[0].key   = winnerKey;
}
```

REFERENCES

ARGE, L. 1995. The buffer tree: A new technique for optimal I/O-algorithms. In *4th WADS*, Number 955 in LNCS (1995), pp. 334–345. Springer.

BARVE, R. D., GROVE, E. F., AND VITTER, J. S. 1997. Simple randomized mergesort on parallel disks. *Parallel Computing 23*, 4, 601–631.

BRENGEL, K., CRAUSER, A., MEYER, U., AND FERRAGINA, P. 1999. An experimental study of prioritty queues in external memory. In *3rd International Workshop on Algorithmic Engineering (WAE)* (1999), pp. 345–359. full paper in ACM Journal of Experimental Algorithmics.

BRODAL, G. S. AND KATAJAINEN, J. 1998. Worst-case efficient external-memory priority queues. In *6th Scandinavian Workshop on Algorithm Theory*, Number 1432 in LNCS (1998), pp. 107–118. Springer Verlag, Berlin.

BROWN, R. 1988. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10, 1220–1227.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* McGraw-Hill.

FADEL, R., JAKOBSEN, K. V., KATAJAINEN, J., AND TEUHOLA, J. 1997. External heaps combined with effective buffering. In *4th Australasian Theory Symposium*, Volume 19-2 of *Australian Computer Science Communications* (1997), pp. 72–78. Springer.

FISCHER, M. J. AND PATERSON, M. S. 1994. Fishspear: A priority queue algorithm. *Journal of the ACM 41*, 1, 3–30.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann.

Intel Corporation. 1997. *Intel Archtecture Software Developer's Manual. Volume I: Basic Architecture.* P.O. Box 5937, Denver, CO, 80217-9808, `http://www.intel.com`: Intel Corporation. Ordering Number 243190.

JONES, D. 1986. An empirical comparison of priority-queue and event set implementations. *Communications of the ACM 29*, 4, 300–311.

KELLER, J. 1996. The 21264: A superscalar alpha processor with out-of-order execution. In *Microprocessor Forum* (October 1996).

KNUTH, D. E. 1973. *The Art of Computer Programming — Sorting and Searching*, Volume 3. Addison Wesley.

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

LAMARCA, A. AND LADNER, R. E. 1997. The influence of caches on the performance of sorting. In *8th Symposium on Discrete Algorithm* (1997), pp. 370–379.

MIPS Technologies, Inc. 1998. *R10000 Microprocessor User's Manual* (2.0 ed.). MIPS Technologies, Inc. `http://www.mips.com`.

NEUMANN, J. V. 1945. First draft of a report on the EDVAC. Technical report, University of Pennsylvania.

SANDERS, P. 1999. Accessing multiple sequences through set associative caches. In *ICALP*, Number 1644 in LNCS (1999), pp. 655–664.

Sun Microsystems. 1997. *UltraSPARC-IIi User's Manual.* Sun Microsystems.

VENGROFF, D. E. 1995. *TPIE User Manual and Reference.* Duke University. `http://www.cs.duke.edu/~dev/tpie_home_page.html`.

VITTER, J. S. 1998. External memory algorithms. In *6th European Symposium on Algorithms*, Number 1461 in LNCS (1998), pp. 1–25. Springer.

VITTER, J. S. AND SHRIVER, E. A. M. 1994. Algorithms for parallel memory I: Two level memories. *Algorithmica 12*, 2–3, 110–147.

WEGENER, I. 1993. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science 118*, 1 (Sept.), 81–98.

WEGNER, L. M. AND TEUHOLA, J. I. 1989. The external heapsort. *IEEE Transactions on Software Engineering 15*, 7 (July), 9–925.