Alcom-FT Technical Report Series
ALCOMFT-TR-03-101

Engineering a Cache-Oblivious Sorting Algorithm*

Gerth Stølting Brodal[†,‡] Rolf Fagerberg[†] Kristoffer Vinther[§]

Abstract

The cache-oblivious model of computation is a two-level memory model with the assumption that the parameters of the model are unknown to the algorithms. A consequence of this assumption is that an algorithm efficient in the cache oblivious model is automatically efficient in a multi-level memory model.

Since the introduction of the cache-oblivious model by Frigo et al. in 1999, a number of algorithms and data structures in the model has been proposed and analyzed. However, less attention has been given to whether the nice theoretical proporities of cache-oblivious algorithms carry over into practice.

This paper is an algorithmic engineering study of cache-oblivious sorting. We investigate a number of implementation issues and parameters choices for the cache-oblivious sorting algorithm Lazy Funnelsort by empirical methods, and compare the final algorithm with Quicksort, the established standard for comparison based sorting, as well as with recent cache-aware proposals.

The main result is a carefully implemented cache-oblivious sorting algorithm, which we compare to the best implementation of Quicksort we can find, and find that it competes very well for input residing in RAM, and outperforms Quicksort for input on disk.

1 Introduction

Modern computers contain a hierarchy of memory levels, with each level acting as a cache for the next. Typical components of the memory hierarchy are: registers, level 1 cache, level 2 cache, level 3 cache, main memory, and disk. The time for accessing a level increases for each new level (most dramatically when going from main memory to disk), making the cost of a memory access depend highly on what is the current lowest memory level containing the element accessed.

As a consequence, the memory access pattern of an algorithm has a major influence on its running time in practice. Since classic asymptotic analysis of algorithms in the RAM model is unable to capture this, a number of more elaborate models for analysis have been proposed. The most widely used of these is the I/O model introduced by of Aggarwal and Vitter [2] in 1988, which assumes a memory hierarchy containing two levels, the lower level having size M and the transfer between the two levels taking place in blocks of B consecutive elements. The cost of the computation is the number of blocks transferred.

*This work is based on the M.Sc. thesis of the third author [29].

†BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf}@brics.dk. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

‡Supported by the Carlsberg Foundation (contract number ANS-0257/20).

§Systematic Software Engineering A/S, Søren Frichs Vej 39, DK-8000 Århus C, Denmark. E-mail: kv@brics.dk.

The strength of the I/O model is that it captures part of the memory hierarchy, while being sufficiently simple to make analysis of algorithms feasible. In particular, it adequately models the situation where the memory transfer between two levels of the memory hierarchy dominates the running time, which is often the case when the size of the data significantly exceeds the size of main memory.

By now, a large number of results for the I/O model exists—see the surveys by Arge [3] and Vitter [30]. Among the fundamental facts are that in the I/O model, comparison based sorting takes $\Theta(\mathrm{Sort}_{M,B}(N))$ I/Os in the worst case, where $\mathrm{Sort}_{M,B}(N) = \frac{N}{B}\log_{M/B}\frac{N}{B}$.

More elaborate models for multi-level memory hierarchies have been proposed ([30, Section 2.3] gives an overview), but fewer analyses of algorithms have been done. For these models, as for the I/O model of Aggarwal and Vitter, algorithms are assumed to know the characteristics of the memory hierarchy.

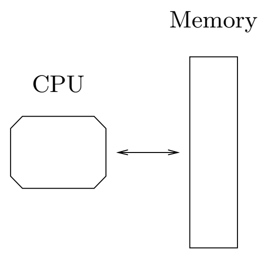


Figure 1: The RAM model

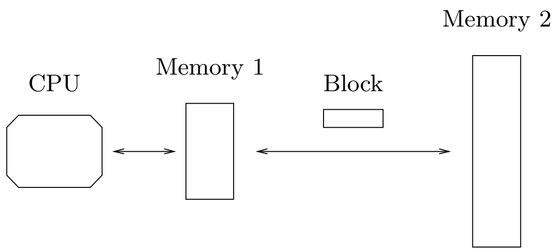


Figure 2: The I/O model

Recently, the concept of *cache-oblivious* algorithms was introduced by Frigo et al. [19]. In essence, this designates algorithms formulated in the RAM model, but analyzed in the I/O model for arbitrary block size B and memory size M. I/Os are assumed to be performed automatically by an offline optimal cache replacement strategy. This seemingly simple change has significant consequences: since the analysis holds for any block and memory size, it holds for *all* levels of a multi-level memory hierarchy (see [19] for details). In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized to each level automatically. Thus, the cache-oblivious model in a elegant way combines the simplicity of the I/O-model with a coverage of the entire memory hierarchy. An additional benefit is that the characteristics of the memory hierarchy do not need to be known, and do not need to be hardwired into the algorithm for the analysis to hold. This increases the algorithms portability (a benefit for e.g. software libraries), and its robustness against changing memory resources on machines running multiple processes.

In 1999, Frigo et al. introduced the concept of cache-obliviousness, and presented optimal cache-oblivious algorithms for matrix transposition, FFT, and sorting [19], and also gave a proposal for static search trees [25] with search cost matching that of standard (cache-aware) B-trees [6]. Since then, quite a number of results for the model have appeared, including the following: Bender et al. [11] gave a proposal for cache-oblivious dynamic search trees with search cost matching B-trees. Simpler cache-oblivious search trees with complexities matching that of [11] were presented in [12, 17, 26], and a variant with worst case bounds for updates appear in [8]. Cache-oblivious algorithms have been given for problems in computational geometry [1, 8, 14], for scanning dynamic sets [7], for layout of static trees [9], and for partial persistence [8]. Cache-oblivious priority queues have been developed in [4, 15], which in turn gives rise to several cache-oblivious graph algorithms [4].

Some of these results, in particular those involving sorting and algorithms to which sorting reduces, such as priority queues, are proved under the assumption $M \geq B^2$, which is also known as the *tall cache assumption*. In particular, this applies to the Funnelsort algorithm of Frigo et al. [19]. A variant termed Lazy Funnelsort [14] works under the weaker tall cache assumption $M \geq B^{1+\varepsilon}$ for any fixed $\varepsilon > 0$, at the cost of a $1/\varepsilon$ factor compared to the optimal sorting bound $\Theta(\mathrm{Sort}_{M,B}(N))$ for the case $M \gg B^{1+\varepsilon}$.

Recently, it was shown [16] that a tall cache assumption is necessary for cache-oblivious comparison based sorting algorithms, in the sense that the trade-off attained by Lazy Funnelsort between strength of assumption and cost for the for the case $M \gg B^{1+\varepsilon}$ is best possible. This demonstrates a separation in power between the I/O model and the cache-oblivious model for the problems of comparison based sorting. Separations have also been shown for the problems of permuting [16] and of comparison based searching [10].

In contrast to the abundance of theoretical results described above, empirical evaluations of the merits of cache-obliviousness are more scarce. Existing results have focused on basic matrix algorithms [19], and search trees [17, 23, 26]. Although a bit tentative, they conclude that in these areas, the efficiency of cache-oblivious algorithms lies between that of classic RAM-algorithms and that of algorithms exploiting knowledge about the specific memory hierarchy present (often termed cache-aware algorithms).

In this paper, we investigate the practical value of cache-oblivious methods in the area of sorting. We focus on the Lazy Funnelsort algorithm, since we believe it to have the biggest potential for an efficient implementation among the current proposals for I/O-optimal cache-oblivious sorting algorithms. We explore a number of implementation issues and parameters choices for the cache-oblivious sorting algorithm Lazy Funnelsort, and settle the best choices through experiments. We then compare the final algorithm with tuned versions of Quicksort, which is generally acknowledged to be the fastest all-round comparison based sorting algorithm, as well as with recent cache-aware proposals.

The end result is a carefully implemented cache-oblivious sorting algorithm, which our experiments show can be faster than the best Quicksort implementation we can find, already for input sizes well within the limits of RAM. It also outperforms the recent cache-aware implementations included in the test. On disk the difference is even more pronounced regarding Quicksort, whereas it is slower than a careful implementation of multiway Mergesort such as TPIE [18].

These findings support—and extend to the area of sorting—the conclusion of the previous empirical results on cache-obliviousness, namely that cache-oblivous methods really can lead to actual performance gains over classic algorithms developed in the RAM-model. The gains may not always be as large as for algorithms tuned to the specific parameters of the levels of the memory hierarchy currently dominating the running time, but on the other hand appear more robust, applying to several levels of the memory hierarchy simultaneously.

One observation of independent interest made in this paper is that for the main building block of Funnelsort, namely the k-merger, there is no need for a specific memory layout (contrary to its previous descriptions [14, 19]) for its analysis to hold. Thus, the central feature of the k-merger definition is the sizes of its buffers, and does not include its layout in memory.

The rest of this paper is organized as follows: In Section 2, we describe Lazy Funnelsort. In Section 3, we describe our experimental setup. In Section 4, we develop our optimized implementation of Funnelsort, and in Section 5, we compare it experimentally to a collection of existing efficient sorting algorithms. In Section 6, we sum up our findings.

2 Funnelsort

Three algorithms for cache-oblivious sorting have been proposed so far: Funnelsort [19], its variant Lazy Funnelsort [14], and a distribution based algorithm [19].

These all have the same optimal bound $\frac{N}{B}\log_{M/B}\frac{N}{B}$ on the number of I/Os performed, but have rather different structural complexity, with Lazy Funnelssort being the simplest. As simplicity of description often translates into smaller and more efficient code (for algorithms of same asymptotic complexity), we find the Lazy Funnelssort algorithm the most promising with respect to practical efficiency, and we choose is as the basis for our study of the practical feasibility of cache-oblivious sorting. In this section, we review the algorithm, and give a observation which further simplifies the algorithm. For the full details, see [14].

The algorithm is based on *binary mergers*. A binary merger takes as input two sorted streams of elements and delivers as output the sorted stream formed by merging of these. One merge step moves an element from the head of one of the input streams to the tail of the output stream. The heads of the input streams and the tail of the output stream reside in *buffers* holding a limited number of elements. A buffer is simply an array of elements, plus fields storing the capacity of the buffer and pointers to the first and last elements in the buffer. Binary mergers can be combined to *binary merge trees* by letting the output buffer of one merger be an input buffer of another—in other words, binary merge trees are binary trees with mergers at the nodes and buffers at the edges. The leaves of the tree contain the streams to be merged.

An *invocation* of a merger is a recursive procedure which performs merge steps until its output buffer is full (or both input streams are exhausted). If during the invocation an input buffer gets empty (but the corresponding stream is not exhausted), the input buffer is recursively filled by an invocation of the merger having this buffer as its output buffer. If both input streams of a merger get exhausted, the corresponding output stream is marked as exhausted. The procedure (except for the issue of exhaustion) is shown in Figure 3 as the procedure $\textsc{Fill}(v)$. A single invocation $\textsc{Fill}(r)$ on the root r of the merge tree will produce a stream which is the merge of the streams at the leaves of the tree.

One particular merge tree is the k-merger. For k a power of two, a k-merger is a perfect binary tree of $k-1$ binary mergers with appropriate sized buffers on the edges, k input streams, and an output buffer at the root of size k^d, for a parameter $d > 1$. A 16-merger is illustrated in Figure 4.

The sizes of the buffers are defined recursively: Let the *top tree* be the subtree consisting of all nodes of depth at most $\lceil i/2 \rceil$, and let the subtrees rooted by nodes at depth $\lceil i/2 \rceil + 1$ be the *bottom trees*. The edges between nodes at depth $\lceil i/2 \rceil$ and depth $\lceil i/2 \rceil + 1$ have associated buffers of size $\alpha\lceil d^{3/2} \rceil$, where α is a positive parameter,[1] and the sizes of the remaining buffers are defined by recursion on the top tree and the bottom trees.

[1] The parameter α is introduced in this paper for tuning purposes.

```
Procedure FILL(v)
    while v's output buffer is not full
        if left input buffer empty
            FILL(left child of v)
        if right input buffer empty
            FILL(right child of v)
        perform one merge step
```

Figure 3: The merging algorithm

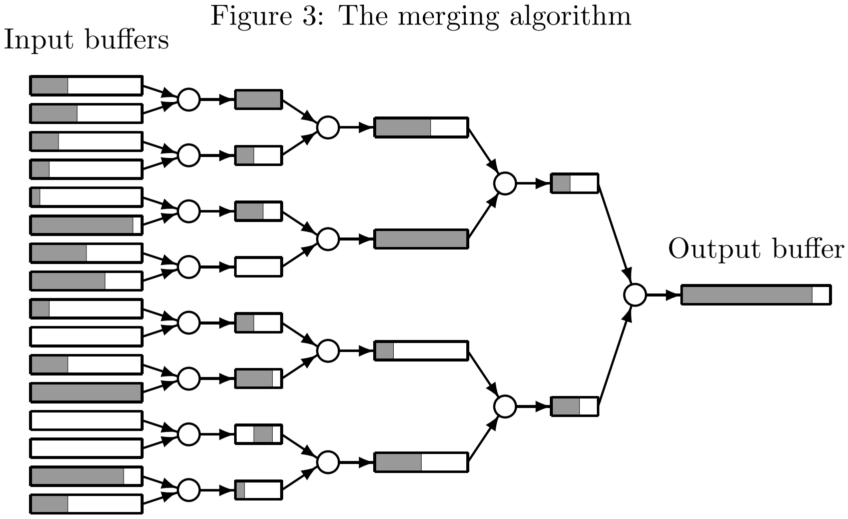


Figure 4: A 16-merger consisting of 15 binary mergers. Shaded regions are the occupied parts of the buffers.

In the descriptions in [14, 19], a k-merger is also laid out recursively in memory (according to the so-called van Emde Boas layout [25]), in order to achieve I/O efficiency. We observe in this paper that this is not necessary: In the proof of Lemma 1 in [14], the central idea is to follow the recursive definition down to a specific size $\bar{k}$ of trees, and then consider the number of I/Os for loading this $\bar{k}$-merger and one block for each of its output streams into memory. However, this price is not (except for constant factors) changed if we for each of the $\bar{k}-1$ nodes have to load one entire block holding the node, and one block for each of the nodes input and output buffers. From this follows that the proof holds true, no matter how the k-merger is laid out. Hence, the crux of the definition of the k-merger lies entirely in the definition of the sizes of the buffers, and does not include the van Emde Boas layout.

To actually sort N elements, the algorithm recursively sorts $N^{1/d}$ segments of size $N^{1-1/d}$ of the input and then merges these using an $N^{1/d}$-merger. For a proof that this is an I/O optimal algorithm, see [14, 19].

3 Methodology

As said, our goal is first to develop a good implementation of Funnelsort by finding good choices for design options and parameter values through empirical investigation, and then to compare its efficiency to that of Quicksort—the established standard for comparison based sorting algorithms—as well as that of recent cache-aware proposals.

To ensure robustness of the conclusions, we perform all experiments on three rather different architectures, namely Pentium 4, Pentium III, and MIPS 10000. These are representatives of the modern CISC, the classic CISC, and the RISC type of computer architecture, respectively. The specifications of the three machines are as follows.

	Pentium 4	*Pentium III*	*MIPS 10000*
Architecture type	Modern CISC	Classic CISC	RISC
Operation system	Linux v. 2.4.18	Linux v. 2.4.18	IRIX v. 6.5
Clock rate	2400MHz	800MHz	175MHz
Address space	32 bit	32 bit	64 bit
Pipeline stages	20	12	6
L1 data cache size	8 KB	16 KB	32 KB
L1 line size	128 B	32 B	32 B
L1 associativity	4-way	4-way	2-way
L2 cache size	512 KB	256 KB	1024 KB
L2 line size	128 B	32 B	32 B
L2 associativity	8-way	4-way	2-way
TLB entries	128	64	64
TLB associativity	full	4-way	64-way
TLB miss handling	hardware	hardware	software
RAM size	512 MB	256 MB	128 MB

Our programs are written in C++ and compiled by GCC v. 3.1.1 (Pentiums 4 and III) or MIPS Pro v. 7.3.1 (MIPS 10000), with full optimization.

We use three data types: integers, records containing one integer and one pointer, and records of 100 bytes. The first type is commonly used in experimental papers, but we do not find it particularly realistic, as keys normally have associated information. The second type models sorting small records directly, as well as key-sorting of large records. The third type models sorting medium sized records directly, and is the data type used in the Datamation Benchmark [20] originating from the database community.

We mainly consider uniformly distributed keys, but also try skewed inputs such as almost sorted data, and data with few distinct key values, to ensure robustness of the conclusions. To keep the experiments during the engineering phase (Section 4) tolerable in number, we only use the pair data type and the uniform distribution in these, believing that tuning based on these will transfer to other situations.

We use the `drand48` family of C library functions for generation of random values. Our performance metric is wall clock time, as measured by the `gettimeofday` C library function.

We keep the code for the different implementation options tested in the engineering phase as similar as possible, even though this genericity entails some overhead. After judging what is the best choices of these options, we implement a clean version of the resulting algorithm, and use this in the final comparison against existing sorting algorithms.

Due to space limitations, we in this paper mainly sum up our findings, and show only few plots of experimental data. The full set of plots (close to hundred) can be found in [29].

4 Engineering Lazy Funnelsort

We consider a number of design and parameter choices for our implementation of Lazy Funnelsort. We group them as indicated by the following subsections. To keep the number of experiments within realistic limits, we settle the choices one by one, in the order presented here. We test each particular question by experiments exercising only parts of the implementation, and/or by fixing the remaining choices at hopefully reasonable values while varying the parameter under investigation.

4.1 k-Merger Structure

As noted in section 2, no particular layout is needed for the analysis of Lazy Funnelsort to hold. However, *some* layout has to be chosen, and the choice could affect the running time. We consider BFS, DFS, and vEB layout. We also consider having a merger node stored along with its output buffer, or storing nodes and buffers separately (each part having the same layout).

The usual tree navigation method is by pointers. However, for the three layouts above, implicit navigation using arithmetic on node indices is possible—this is well-known for BFS [31], and arithmitic expressions for DFS and vEB layouts can be found in [17]. Implicit navigation saves space at the cost of more CPU cycles per navigation step. We consider both pointer based and implicit navigation.

We try two coding styles for the invocation of a merger, namely the straight-forward recursive implementation, and an iterative version. To control the forming of the layouts, we make our own allocation function, which starts by acquiring enough contiguous space to hold the entire merger. We test the overhead of this by also trying out the default allocator in C++. Using this, we have no guarantee that the proper memory layouts are formed, so we only try pointer based navigation in these cases.

Experiments: We test all combinations of the choices described above, except for a few infeasibel ones (e.g. implicit navigation with the default allocator), giving a total of 28 experiments on each of the three machines. One experiment consists of merging k streams of k^2 elements in a k-merger with $z = 2$, $\alpha = 1$, and $d = 2$. For each choice, we for values of k in $[15; 270]$ measure the time for $\lceil 20,000,000/k^3 \rceil$ such mergings.

Results: The best combination on all architectures is recursive invocation of a pointer based vEB layout with nodes and buffers separate, allocated by the standard allocator. The time used for the slowest combination is up to 65% larger, and the difference is biggest on the Pentium 4. The largest gain occurs by choosing the recursive invocation over the iterative, and this gain is most pronounced on the Pentium 4, which also has the most sophisticated architecture (it e.g. has a special return address stack holding the address of the next instruction to be fetched after returning from a function call, for its immediate execution). The vEB layout ensures around 10% reduction in time, which shows that the the spatial locality of the layout is not entirely without influence in practice, despite its lack of influence on the analysis. The implicit vEB layout is slower than its pointer based version, but less so on the Pentium 4, which also is the fastest of the processors, and can execute the most CPU cycles per memory access.

4.2 Tuning the Basic Mergers

The "inner loop" in the Lazy Funnelsort algorithm is the code performing the merge step in the nodes of the k-merger. We explore several ideas for efficient implementation of this code. One idea tested is to compute the minimum of the number of elements left in either input buffer and the space left in the output buffer. Merging can proceed for at least that that many steps without checking the state of the buffers, and this moves one branch out of the core merging loop. We also try several hybrids of this idea and the basic merger.

This idea will not be a gain (rather, the minimum computation will constitute an overhead) in situations where one input buffer stays small for many merge steps. For this reason, we also implement the optimal merging algorithm of Hwang and Lin [21, 22], which has higher overhead, but is an asymptotically improvement when merging sorted list of very different size. To counteract its overhead, we also try a hybrid solution which invokes it only when the contents of the input buffers are skewed in size.

Experiments: We run the same experiment as in Section 4.1. The of values of α and d influence the sizes of the smallest buffers in the merger. These smallest buffers occur on every second level, so any node has one of these as either input or output buffer, making this size affect the heuristics above. For this reason, we repeat the experiment for (α, d) equal to $(1, 3)$, $(4, 2.5)$, and $(16, 1.5)$. These have smallest buffer sizes of 8, 23, and 45, respectively.

Results: The Hwang-Lin algorithm has, as expected, a large overhead (a factor of three for the non-hybrid version). Somewhat to our surprise, the heuristic calculating minimum sizes is not competitive, being between 15% and 45% slower than the fastest, (except on the MIPS 10000 architecture, where the differences between heuristics are less pronounced). Several hybrids fare better, but the straight-forward solution is consistently the winner in all experiments. We interpret this as the branch prediction of the CPUs being as efficient as explicit hand-coding for exploiting predictability in the branches in this code (all branches, except the result of the comparison of the heads of the input buffers, are rather predictable). Thus, hand-coding just constitutes overhead.

4.3 Degree of Basic Mergers

There is no need for the k-merger to be a binary tree. If we for instance base it on four-ary basic mergers, we effectively remove every second level of the tree. This means less element movement and less tree navigation. In particular, a reduction in data movement seems promising—part of Quicksorts speed can be attributed to the fact that for random input, only about every other element is moved on each level in the recursion, whereas e.g. binary Mergesort moves all elements at each level. The price to pay is more CPU steps per merge step, and code complication due to the increase in number of input buffers that can be exhausted.

Based on considerations of expected register use, element movements, and number of branches, we try several different ways of implementing multi-way mergers using sequential comparison of the front elements in the input buffers. We also try a heap-like approach using looser trees [22], which in a previous study by Sanders [27] of priority queues in RAM have proved themselves efficient. In total, seven proposals for multi-way mergers are implemented.

Experiments: We test the seven implementations in a 120-merger with $(\alpha, d) = (16, 2)$, and measure the time for eight mergings of 1,728,000 elements each. The test is run for degrees $z = 2, 3, 4, \ldots, 9$. For comparison, we also include the binary mergers from the last set of experiments.

Results: All implementations except the looser tree show the same behaviour: As z goes from 2 to 9, the time first decreases, and then increases again, with minimum attained around 4 or 5. The maximum is 40–65% slower than the fastest. Since the number of levels for elements to move through evolves as $1/\log(z)$, while the number of comparisons for each level evolves as z, a likely explanation is that there is an initial positive effect due to decrease in element movements, which soon is overtaken by increase in instruction count per level. The looser trees show decrease only in running time for increasing z, consistent with the fact that the number of comparisons per element for a traversal of the merger is the same for all values of z, but the number levels, and hence data movements, evolves as $1/\log(z)$. Unfortunately, the running time starts out twice as large as for the remaining implementations for $z = 2$, and barely reaches them at $z = 9$. Apparently, the overhead is too large to make looser trees competitive in this setting. The plain binary mergers compete well, but are beaten by around 10% by the fastest four- or five-ary mergers. All these findings are rather consistent across the three architectures.

4.4 Merger Caching

In the outer recursion of Funnesort, the same size k-merger is used for all invocations on the same level of the recursion. A natural optimization would be to precompute these sizes and construct the needed k-mergers once for each size. These mergers are then reset each time they are used.

Experiments: We use the Lazy Funnelsort algorithm with $(\alpha, d, z) = (4, 2.5, 2)$, straight-forward implementation of binary basic mergers, and a switch to `std::sort`, the STL implementation of Quicksort, for sizes below $\alpha z^d = 23$. We sort instances ranging in size from 5,000,000 to 200,000,000 elements.

Results: On all architectures, merger caching gave a 3–5% speed-up.

4.5 Base Sorting Algorithm

Like any recursive algorithm, the base case in Lazy Funnelsort is handled specially. As a naturally limit, we require all k-mergers to have height at least two—this will remove a number of special cases in the code constructing the mergers. Therefore, we for input sizes below αz^d switch to another sorting algorithm. Experiments with the sorting algorithms Insertionsort, Selectionsort, Heapsort, Shellsort, and Quicksort (in the form of `std::sort` from the STL library) on input size from 10 to 100 revealed the expected result, namely that `std::sort` (which itself swithes to Insertionsort below size 20), is the fastest for all sizes. We therefore choose this as the sorting algorithm for the base case.

4.6 Parameters α and d

The final choices concerns the parameters α (factor in buffer size expression) and d (main parameter defining the progression of the recursion, in the outer recursion of Funnelsort, as well as in the buffer sizes in the k-merger). These control the buffer sizes, and we investigate their impact on the running time.

Experiments: For values of d between 1.5 and 3 and for values of α between 1 and 40, we measure the running time for sorting inputs of various sizes in RAM.

Results: There is a marked raise in running time when α gets below 10, increasing to a factor of four for $\alpha = 1$. This effect is particularly strong for $d = 1.5$. Smaller α and d give smaller sizes of buffers, and the most likely explanation seems to be that the cost of navigating to and invoking a basic merger is amortized over fewer merge steps when the buffers are smaller. Other than that, the different values of d appears to behave quite similarly. A sensible choice appears to be α around 16, and d around 2.5

5 Evaluating Lazy Funnelsort

In Section 4, we settled the best choices for a number of implementation issues for Lazy Funnelsort. In this section, we investigate the practical merits of the resulting algorithm.

5.1 Competitors

Comparing algorithms with the same asymptotic running time is a delicate matter. Tuning of code can often change the constants involved significantly, which leaves open the question of how to ensure equal levels of engineering in implementations of different algorithms.

Our choice in this paper is to use Quicksort as the main measure stick. Quicksort is known as a very fast general-purpose comparison based algorithm [28], and has long been the standard choice of sorting algorithm in software libraries. Over the last 30 years, many improvements have been suggested and tried, and the amount of practical experience with Quicksort is probably unique among sorting algorithms. It seems reasonable to expect implementations in current libraries to be highly tuned. To further boost confidence in the efficiency of the chosen Quicksort implementation of Quicksort, we start by comparing several widely used library implementations, and choose the best performer as our main competitor. We believe such a comparison will give a good picture of the practical feasibility of cache-oblivious ideas in the area of comparison based sorting.

The implementations we consider are `std::sort` from the STL library included in the GCC distribution, `std::sort` from the STL library from Dinkumware[2] included with the Microsoft Visual C++ compiler, an implementation based on a proposal of Bentley and McIlroy [13], and an implementation of our own, incorporating elements from the others and adding a more elaborate choice of pivot element for large instances. These algorithms mainly differ in their partitioning strategies—how meticulously they choose the pivot element, and whether they use two- or three-way partitioning.

To gain further insight, we also compare with implementations of cache-aware sorting algorithms aiming for efficiency in either internal memory or external memory by tunings based on knowledge of the memory hierarchy.

TPIE [18] is a library for external memory computations, and includes highly optimized routines for e.g. scanning and sorting. We choose TPIEs sorting routine `AMI_sort` as representative of sorting algorithms efficient in external memory. The algorithm needs to know the amount of available internal memory, and following suggestions in the TPIEs manual we set it to 192 Mb, which is 50–75% of the physical memory on all machines where it is tested.

Several recent proposals for cache-aware sorting algorithms in internal memory exist, including [5, 24, 32]. LaMarca and Ladner [24] give proposals for better exploiting L1 and L2 cache. Improving on their effort, Arge et al. [5] give proposals using registers better, and Kubricht et al. [32] give variants of the algorithms from [24] taking the effects of TLB (Translation Look-aside Buffers) misses and the low associativity of caches into account.

In this test, we compare against the Mergesort based proposals from [24], as implemented by [32]—we encountered problems with the remaining implementations from [32], and could not get hold of the code from [5] in time for these experiments.

5.2 Experiments

We test the algorithms describe above on inputs of sizes in the entire RAM range, as well as on inputs residing on disk. All experiments are performed on machines with no other users. The

[2] `www.dinkumware.com`

influence of background processes is minimized by running each experiment in internal memory 21 times, and reporting the median. In external memory experiments are rather time consuming, and we run each experiment only once, believing that background processes will have litte impact on these.

Besides the three machines mentioned in Section 3, we in these final experiments also include an AMD Athlon 1330 Mhz processor with 512 Mb of RAM, and an Intel Itanium 2 1100 Mhz 64 bit processor with 3 Gb of RAM.. The methodology is as described in Section 3, except that we have to use another compiler (Intel C++, v. 7) on the Itanium, and an older version (2.96) of the GCC compiler for the TPIE code (hence, we can not run the TPIE implementation on the Itanium).

5.3 Results

The plots described in this section can all be found in Appendix A. The y-axis shows wall time divided by $n \log n$ in seconds, and the x-axis is $logn$.

The comparison of Quicksort implementations showed that three contestants ran pretty close, with the GCC implementation as the overall winner. It uses a compact two-way partitioning scheme, and simplicity of code here seems to pay off. It is closely followed by our own implementation (denoted `Mix`) in the plots, and the Bentley-McIlroy implementation (denoted `BI`). The implementation from the Dinkumware STL library lags behind, probably due to its rather involved three-way partitioning routine. We choose the GCC implementation as the Quicksort to participate in the remaining experiments.

For the experiments in RAM, we see that for the smallest sizes, Lazy Funnelsort looses to GCC Quicksort (by some 50% on average), but on three architectures gains as n grows, ending up as the winner for the largest instances in RAM. The two architectures where GCC keeps its lead are the MIPS 10000, which has a very slow CPU, and the Pentium 4, which features the PC800 bus, which decreases the access time to RAM. This can be interpreted as on these two architectures, CPU cycles, and not cache effects are dominating the running time, and on architectures where this is not the case, the theoretically better cache performance of Funnelsort actually shows through in practice. at least for this tuned implementation of the algorithm. The two cache-aware implementations are not competitive on any of the architectures.

For the experiments on disk, TPIE (denoted `amisort`) is the clear winner. It is highly optimized for external memory, and we suspect in particular that its use of duble-buffering—something which seems hard to transfer to a cache-oblivious setting—gives it an unbeatable advantage[3]. However, Funnelsort comes in as a second, and outperforms GCC quite clearly. The gain over GCC seems to grow as n grows larger.

Due to lack of space, we here only show plots for uniformly distributed data of the pair type. The results for the other types and distributions discussed in Section sec:methodology are quite similar, and can be found in [29].

6 Conclusion

Through a careful engineering effort, we have developed an implementation of Lazy Funnelsort, with a instruction count low enough for its memory optimal behaviour to show through on disk as well as in RAM, for architectures where sorting is not CPU bound.

[3]The TPIE sorting routine sorts one run while loading the next from disk, thus parallelizing CPU work and I/Os.

This demonstrates, somewhat to our surprise, that the overhead involved in being cache-oblivious can be small enough for the nice theoretical properties to transfer into practical advantages.

References

[1] P. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. On cache-oblivious multidimensional range searching. In *Proc. 19th ACM Symposium on Computational Geometry*, 2003. To appear.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[3] L. Arge. External memory data structures. In *Proc. 9th Annual European Symposium on Algorithms (ESA)*, volume 2161 of *LNCS*, pages 1–29. Springer, 2001.

[4] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing*, pages 268–276. ACM Press, 2002.

[5] L. Arge, J. Chase, J. Vitter, and R. Wickremesinghe. Efficient sorting using registers and caches. *ACM Journal of Experimental Algorithmics*, 7(9), 2002.

[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[7] M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 139–151. Springer, 2002.

[8] M. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 195–207. Springer, 2002.

[9] M. Bender, E. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 165–173. Springer, 2002.

[10] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The cost of cache-oblivious searching. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003. To appear.

[11] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Ann. Symp. on Foundations of Computer Science*, pages 399–409. IEEE Computer Society Press, 2000.

[12] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 29–39, 2002.

[13] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software–Practice and Experience*, 23(1):1249–1265, 1993.

[14] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 426–438. Springer, 2002.

[15] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *LNCS*, pages 219–228. Springer, 2002.

[16] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 307–315, 2003.

[17] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002.

[18] Department of Computer Science, Duke University. TPIE: a transparent parallel I/O environment. WWW page, `http://www.cs.duke.edu/TPIE/`, 2002.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.

[20] J. Gray. Sort benchmark home page. WWW page, `http://research.microsoft.com/barc/SortBenchmark/`, 2003.

[21] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[22] D. E. Knuth. *The Art of Computer Programming, Vol 3, Sorting and Searching.* Addison-Wesley, Reading, USA, 2 edition, 1998.

[23] R. E. Ladner, R. Fortna, and B.-H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics*, volume 2547 of *LNCS*, pages 78–92. Springer, 2002.

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

[25] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[26] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. 5th Int. Workshop on Algorithm Engineering (WAE)*, volume 2141, pages 67–78. Springer, 2001.

[27] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5(7), 2000.

[28] R. Sedgewick. *Algorithms in C: Parts 1–4: Fundamentals, data structures, sorting, searching.* Addison-Wesley, Reading, MA, USA, 1998.

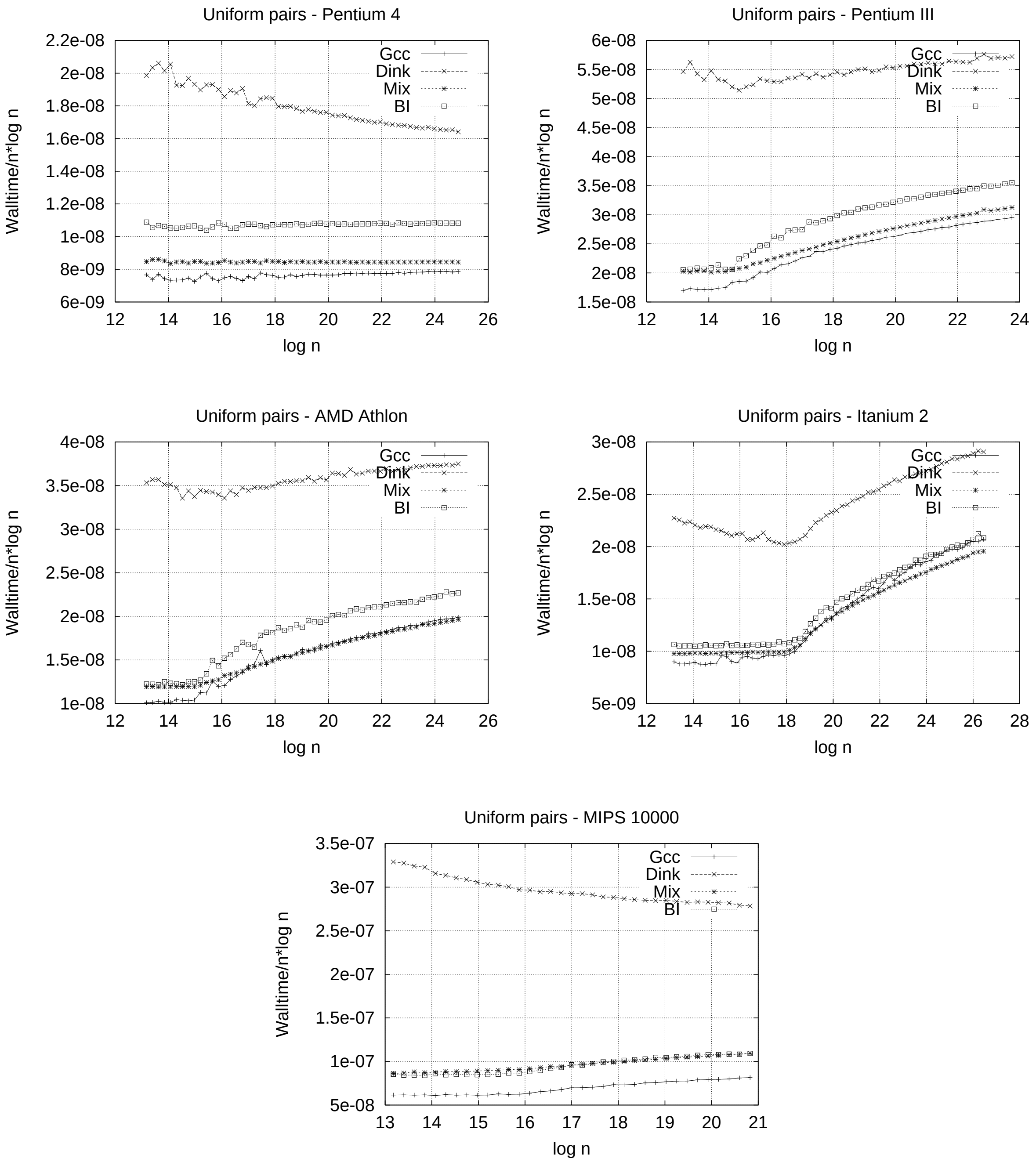
[29] K. Vinther. Engineering cache-oblivious sorting algorithms. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, 2003. Available online at `http://www.daimi.au.dk/~kv/thesis/`.

[30] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

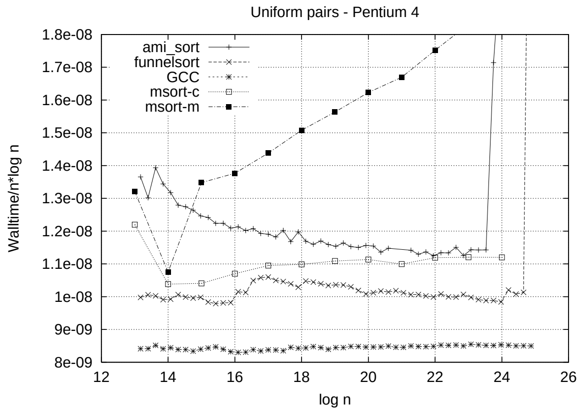
[31] J. W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7:347–348, 1964.

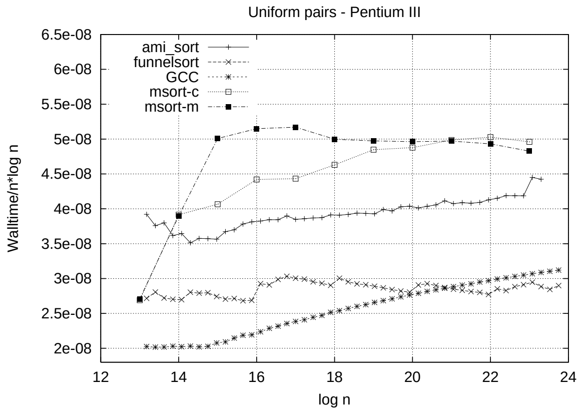
[32] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal of Experimental Algorithmics*, 5(3), 2000.

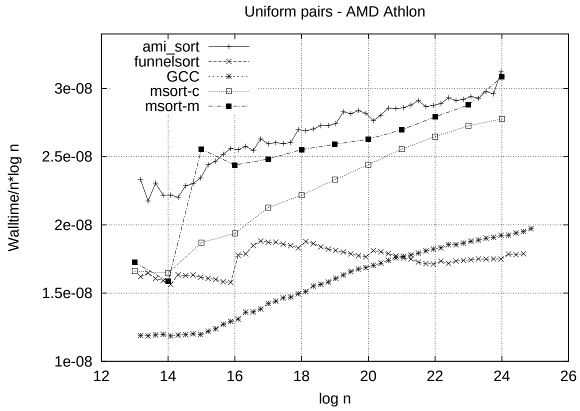
A Plots

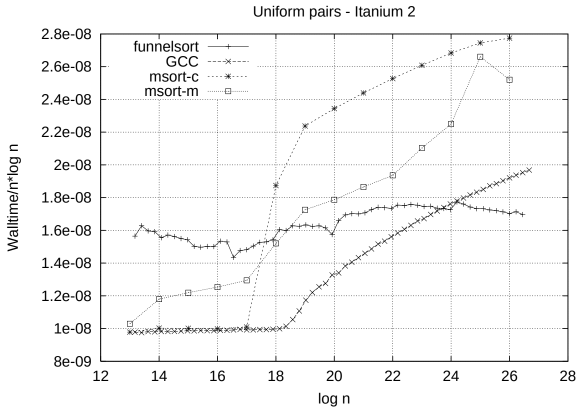


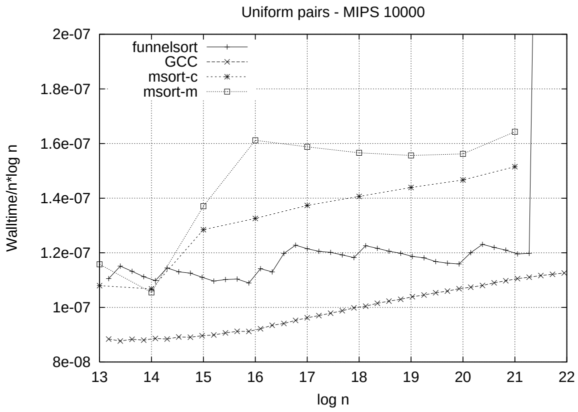
Comparison of Quicksort implementations



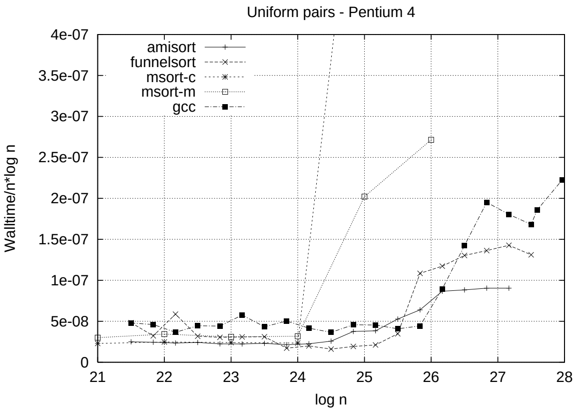


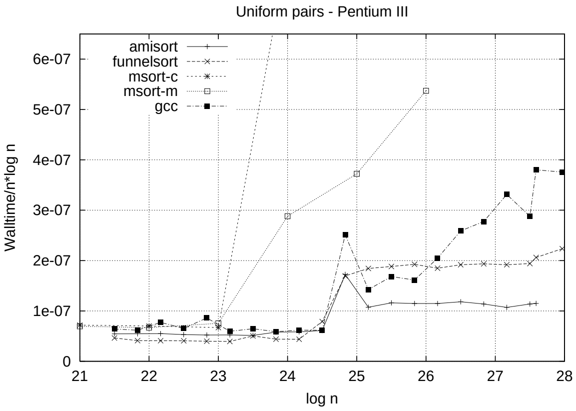


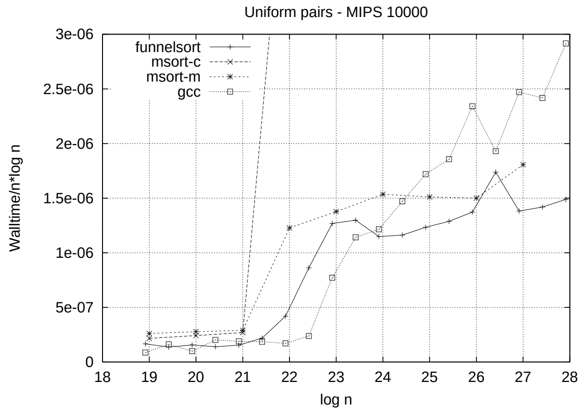




Results for inputs in RAM







Results for inputs on disk