Engineering a Cache-Oblivious Sorting Algorithm

GERTH STØLTING BRODAL
University of Aarhus
ROLF FAGERBERG
University of Southern Denmark, Odense
and
KRISTOFFER VINTHER
Systematic Software

This paper is an algorithmic engineering study of cache-oblivious sorting. We investigate by empirical methods a number of implementation issues and parameter choices for the cache-oblivious sorting algorithm Lazy Funnelsort, 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 our experiments show can be faster than the best Quicksort implementation we are able to find, already for input sizes well within the limits of RAM. It is also at least as fast as the recent cache-aware implementations included in the test. On disk the difference is even more pronounced regarding Quicksort and the cache-aware algorithms, whereas the algorithm is slower than a careful implementation of multiway Mergesort such as TPIE.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and Searching*; B.3.2 [**Memory Structures**]: Design styles—*Cache memories*; *Virtual memory*; E.5 [**Files**]: —*Sorting/searching*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Cache oblivious algorithms, funnelsort, quicksort

Gerth Stølting Brodal, BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: gerth@brics.dk. Supported by the Carlsberg Foundation (contract number ANS-0257/20). Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
Rolf Fagerberg, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark. E-mail: `rolf@imada.sdu.dk`. Work done while at University of Aarhus. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
Kristoffer Vinther, Systematic Software Engineering A/S, Søren Frichs Vej 39, DK-8000 Århus C, Denmark. E-mail: `academia@kristoffer.vinther.name`. Work done while at University of Aarhus.

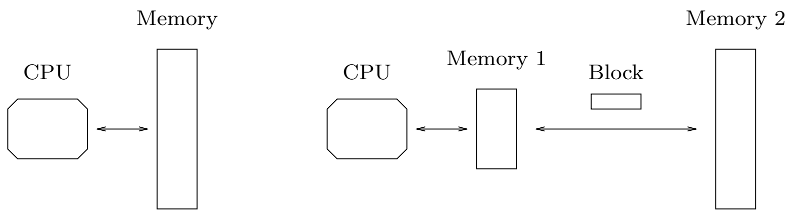


Fig. 1. The RAM and the I/O models.

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 it takes to access a level increases for each new level, most dramatically when going from main memory to disk. Consequently, the cost of a memory access depends 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 analyses of algorithms in the RAM model are 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 in [Aggarwal and Vitter 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. Figure 1 illustrates the RAM and the I/O models.

The strength of the I/O model is that it captures part of the memory hierarchy while being simple enough to make analyses 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 exceeds the size of main memory.

By now, a large number of results for the I/O model exist—see the surveys [Arge 2001] and [Vitter 2001]. Among the fundamental facts are that in the I/O model, comparison-based sorting takes $\Theta(\text{Sort}_{M,B}(N))$ I/Os in the worst case, where $\text{Sort}_{M,B}(N) = \frac{N}{B}\log_{M/B}\frac{N}{B}$.

More elaborate models for multi-level memory hierarchies have been proposed ([Vitter 2001, 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.

Recently, the concept of *cache-oblivious* algorithms was introduced in [Frigo et al. 1999]. 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 [Frigo et al. 1999] for details). In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized to all levels automatically. Thus, the cache-oblivious model combines in an elegant way the simplicity of the two-level 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 portability of the algorithm (a benefit for e.g. software libraries), as well as 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 [Frigo et al. 1999], and also gave a proposal for static search trees [Prokop 1999] with search cost matching that of standard (cache-aware) B-trees [Bayer and McCreight 1972]. Since then, quite a number of results for the model have appeared, including the following: Cache-oblivious dynamic search trees with search cost matching B-trees was given in [Bender et al. 2000]. Simpler cache-oblivious search trees with complexities matching that of [Bender et al. 2000] were presented in [Bender et al. 2002; Brodal et al. 2002; Rahman et al. 2001], and a variant with worst case bounds for updates appeared in [Bender et al. 2002]. Further cache-oblivious dictionary structures have been given in [Arge et al. 2005; Bender et al. 2006; Bender et al. 2005; Brodal and Fagerberg 2006; Franceschini and Grossi 2003a; 2003b], and further cache-oblivious sorting results in [Brodal and Fagerberg 2002a; Brodal et al. 2005; Fagerberg et al. 2006; Farzan et al. 2005; Franceschini 2004]. Cache-oblivious algorithms have been presented for problems in computational geometry [Agarwal et al. 2003; Arge et al. 2005; Bender et al. 2002; Brodal and Fagerberg 2002a], for scanning dynamic sets [Bender et al. 2002], for layout of static trees [Bender et al. 2002], for search problems on multi-sets [Farzan et al. 2005], for dynamic programming [Chowdhury and Ramachandran 2006], and for partial persistence [Bender et al. 2002]. Cache-oblivious priority queues have been developed in [Arge et al. 2002; Brodal and Fagerberg 2002b], which in turn give rise to several cache-oblivious graph algorithms [Arge et al. 2002]. Other cache-oblivious graph algorithms appear in [Brodal et al. 2004; Chowdhury and Ramachandran 2004; Jampala and Zeh 2005]. For a further overview of cache-oblivious algorithms, see the surveys [Arge et al. 2005; Brodal 2004].

Some of these results, in particular those involving sorting and algorithms to which sorting reduces (e.g. 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 in [Frigo et al. 1999]. A variant termed Lazy Funnelsort [Brodal and Fagerberg 2002a] works under the weaker tall cache assumption $M \geq B^{1+\varepsilon}$ for any fixed $\varepsilon > 0$, at the cost of being a factor $1/\varepsilon$ worse than the optimal sorting bound $\Theta(\mathrm{Sort}_{M,B}(N))$ when $M \gg B^{1+\varepsilon}$.

It has been shown [Brodal and Fagerberg 2003] 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 case $M \gg B^{1+\varepsilon}$ is the best possible. This demonstrates a separation in power between the I/O model and the cache-oblivious model for comparison-based sorting. Separations have also been shown for permuting [Brodal and Fagerberg

2003] and for comparison-based searching [Bender et al. 2003].

Compared to the abundance of theoretical results described above, empirical evaluations of the merits of cache-obliviousness are more scarce. Results exist for areas such as basic matrix algorithms [Frigo et al. 1999], dynamic programming algorithms [Chowdhury and Ramachandran 2006], and search trees [Brodal et al. 2002; Ladner et al. 2002; Rahman et al. 2001]. They conclude that in these areas, the efficiency of cache-oblivious algorithms surpasses classic RAM-algorithms, and competes well with that of algorithms exploiting knowledge about the memory hierarchy parameters.

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 has 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 parameter 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. Note that the I/O cost of Quicksort is $\Theta(\frac{N}{B}\log_2\frac{N}{M})$, which only differs from the optimal bound $\mathrm{Sort}_{M,B}(N)$ by the base of the logarithm.

The main result is a carefully implemented cache-oblivious sorting algorithm, which our experiments show can be faster than the best Quicksort implementation we are able to find, already for input sizes well within the limits of RAM. Also, it is just as fast as the best of the recent cache-aware implementations included in the test. On disk the difference is even more pronounced regarding Quicksort and the cache-aware algorithms, whereas the algorithm is slower than a careful implementation of multiway Mergesort such as TPIE [Department of Computer Science, Duke University 2002].

These findings support—and extend to the area of sorting—the conclusion of the previous empirical results on cache-obliviousness. This conclusion is that cache-oblivious methods can lead to actual performance gains over classic algorithms developed in the RAM-model. The gains may not always entirely match those of the best algorithm tuned to a specific memory hierarchy level, but on the other hand appear to be more robust, applying to several memory hierarchy levels 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 [Brodal and Fagerberg 2002a; Frigo et al. 1999]) for its analysis to hold. Thus, the essential feature of the k-merger definition is the sizes of the buffers, not the 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 summarize our findings.

The work presented is based on the M.Sc. thesis [Vinther 2003].

```
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
```

Fig. 2. The merging algorithm.

2. FUNNELSORT

Five algorithms for cache-oblivious sorting seem to have been proposed so far: Funnelsort [Frigo et al. 1999], its variant Lazy Funnelsort [Brodal and Fagerberg 2002a], a distribution based algorithm [Frigo et al. 1999], an implicit algorithm [Franceschini 2004], and an adaptive sorting algorithm [Brodal et al. 2005].

These all have the same optimal bound $\mathrm{Sort}_{M,B}(N)$ on the number of I/Os performed, but have rather different structural complexity, with Lazy Funnelsort being the simplest. As simplicity of description often translates into smaller and more efficient code (for algorithms of the same asymptotic complexity), we find the Lazy Funnelsort algorithm the most promising with respect to practical efficiency. In this paper we choose it as the basis for our study of the practical feasibility of cache-oblivious sorting. We now review the algorithm briefly, and give an observation which further simplifies it. For the full details of the algorithm, see [Brodal and Fagerberg 2002a].

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 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 two input streams and the tail of the output stream reside in *buffers* holding a limited number of elements. A buffer consists of an array of elements, a field 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 becomes 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 become exhausted, the corresponding output stream is marked as exhausted. The procedure (except for the issue of exhaustion) is shown in Figure 2 as the procedure Fill(v). A single invocation Fill(r) on the root r of the merge tree will merge the streams at the leaves of the tree, provided that the output buffer of the root has size at least the sum of the sizes of these streams.

One particular merge tree is the *k-merger*. A k-merger consists of $k-1$ binary mergers forming a binary tree of optimal height $i = \lceil \log k \rceil$. The tree has k input streams as leaves, an output buffer of size k^d (where $d > 1$ is a parameter) at

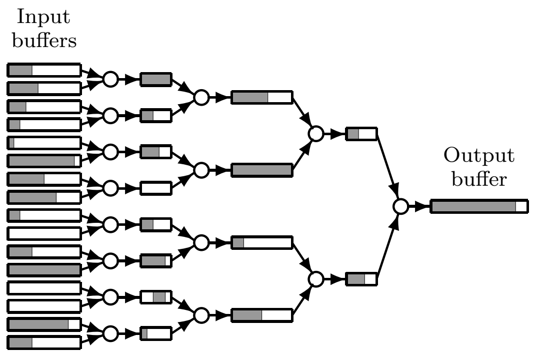


Fig. 3. A 16-merger consisting of 15 binary mergers. Shaded regions are the occupied parts of the buffers.

the root, and buffers (of sizes to be defined below) on the edges. A 16-merger is illustrated in Figure 3.

The sizes of the buffers are defined recursively: Let the depth of a node be one for the root, and one plus the depth of the parent for any other node. 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.

The k-merger structure was defined in [Frigo et al. 1999] for use in Funnelsort. The algorithm FILL(v) described above for invoking a k-merger appeared in [Brodal and Fagerberg 2002a], and is a simplification of the original one.

In the descriptions in [Brodal and Fagerberg 2002a; Frigo et al. 1999], a k-merger is laid out recursively in memory (according to the so-called van Emde Boas layout [Prokop 1999]), in order to achieve I/O efficiency. We observe here that this is not necessary: In the proof of Lemma 1 in [Brodal and Fagerberg 2002a], 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 needed to load 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 of the $\bar{k}$-merger have to load one entire block holding the node, and one block for each of the input and output buffers of the node. From this follows that the proof holds true, no matter how the k-merger is laid out.[2] Hence, the crux of the definition of the k-merger is the sizes of the buffers, not the layout in memory.

To sort N elements, Lazy Funnelsort first 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. A proof that this is an I/O optimal algorithm can be found in [Brodal and Fagerberg 2002a; Frigo et al. 1999].

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

[2]However, the (entire) k-merger should occupy a contiguous segment of memory in order for the complexity proof (Theorem 2 in [Brodal and Fagerberg 2002a]) of Funnelsort itself to be valid.

Table I. The specifications of the machines used in this paper.

	Pentium 4	*Pentium III*	*MIPS 10000*	*AMD Athlon*	*Itanium 2*
Architecture type	Modern CISC	Classic CISC	RISC	Modern CISC	EPIC
Operation system	Linux v. 2.4.18	Linux v. 2.4.18	IRIX v. 6.5	Linux 2.4.18	Linux 2.4.18
Clock rate	2400MHz	800MHz	175MHz	1333 MHz	1137 MHz
Address space	32 bit	32 bit	64 bit	32 bit	64 bit
Pipeline stages	20	12	6	10	8
L1 data cache size	8 KB	16 KB	32 KB	128 KB	32 KB
L1 line size	128 B	32 B	32 B	64 B	64 B
L1 associativity	4-way	4-way	2-way	2-way	4-way
L2 cache size	512 KB	256 KB	1024 KB	256 KB	256 KB
L2 line size	128 B	32 B	32 B	64 B	128 B
L2 associativity	8-way	4-way	2-way	8-way	8-way
TLB entries	128	64	64	40	128
TLB associativity	full	4-way	64-way	4-way	full
TLB miss handling	hardware	hardware	software	hardware	?
RAM size	512 MB	256 MB	128 MB	512 MB	3072 MB

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 to those 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. In the final comparison of algorithms we add the AMD Athlon (a modern CISC architecture) and the Intel Itanium 2 (denoted an EPIC architecture by Intel, for Explicit Parallel Instruction-set Computing) for even larger coverage. The specifications of all five machines[3] used can be seen in Table I.

Our programs are written in C++ and compiled by GCC v. 3.3.2 (Pentiums 4 and III, AMD Athlon), GCC v. 3.1.1 (MIPS 10000), or the Intel C++ compiler v. 7.0 (Itanium 2). We compile using maximal optimization.

We use three element 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 of small records, as well as key-sorting (i.e. sorting of key-pointer pairs, without movement of the actual data) of large records. The third type models sorting of medium sized records, and is the data type used in the Datamation Benchmark [Gray 2003] 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 second data type and the uniform distribution, believing that tuning based on these will transfer to other situations.

We use the `drand48` family of C library functions to generate random values.

[3]Additionally, the Itanium 2 machine has 3072 KB of L3 cache, which is 12-way associative and has a cache line size of 128 B.

Our performance metric is wall clock time, as measured by the `gettimeofday` C library function.

In the engineering phase we keep the code for testing the various implementation parameters as similar as possible, even though this generality could entail some overhead. After judging what are the best choices for these parameters, we implement a clean version of the resulting algorithm, and use this in the final comparison against existing sorting algorithms.

The total number of experiments we perform is rather large. In this paper, we concentrate on summing up our findings, and show only a representative set of plots of experimental data. Further details and a full set of plots (close to a hundred) can be found in [Vinther 2003]. The entire code for the experiments in Sections 4 and 5 is available online at The JEA Research Code Repository [4] and [Vinther 2003]. The code for the experiments in Section 5 (including our final, tuned Lazy Funnelsort implementation) is also published with the current paper.

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 through experiments exercising only parts of the implementation, and/or by fixing the remaining choices at hopefully reasonable values while varying the parameter under investigation. In this section we summarize the results of each set of experiments—the complete set of plots can be found in [Vinther 2003].

Regarding notation: α and d are the parameters from the definition of the k-merger (see Section 2), and z denotes the degree of the basic mergers (see Section 4.3).

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 [Williams 1964], and arithmetic expressions for DFS and vEB layouts can be found in [Brodal et al. 2002]. 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 straightforward recursive implementation, and an iterative version. To control the forming of the layouts we make our own allocation function, which starts by acquiring enough memory to hold the entire merger. We test the efficiency of our allocation function by also trying out the default allocator in C++. Using this, we have no

[4] `http://www.acm.org/jea/repository/`

guarantee that the proper memory layouts are formed, so we only try pointer based navigation in these cases.

4.1.1 *Experiments.* We test all combinations of the choices described above, except for a few infeasible 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 merges.

4.1.2 *Results.* On all architectures the best combination is 1) recursive invocation, 2) pointer based navigation, 3) vEB layout, 4) nodes and buffers laid out separately, and 5) allocation by the standard allocator. The time used for the slowest combination is up to 65% longer, and the difference is biggest on the Pentium 4 architecture. The largest gain occurs by choosing the recursive invocation over the iterative, and this gain is most pronounced on the Pentium 4 architecture, which is also the most sophisticated (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 spatial locality of the layout is not entirely without influence in practice, despite its lack of influence on the asymptotic analysis. The implicit vEB layout is slower than its pointer based version, but less so on the Pentium 4 architecture, which also is the fastest of the processors and most likely the one least strained by complex arithmetic expressions.

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 numbers of elements left in each input buffer and the space left in the output buffer. Merging can proceed for at least that many steps without checking the state of the buffers, thereby eliminating one branch from 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 [Hwang and Lin 1972; Knuth 1998], which has higher overhead, but is an asymptotic improvement when merging sorted lists of very different sizes. To counteract its overhead, we also try a hybrid solution which invokes the Hwang-Lin algorithm only when the contents of the input buffers are skewed in size.

4.2.1 *Experiments.* We run the same experiment as in Section 4.1. The values of α and d influence the sizes of the smallest buffers in the merger. These smallest buffers occur on every second level of the merger, so any node has one of these as either input or output buffer, making their 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.

4.2.2 *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 that calculates 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-way basic mergers, we effectively remove every other level of the tree. This means less element movement and less tree navigation. In particular, a reduction in data movement seems promising—part of Quicksort's 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 on 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 [Knuth 1998], which proved efficient in a previous study by Sanders [Sanders 2000] of priority queues in RAM. In total, seven proposals for multi-way mergers are implemented.

4.3.1 *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.

4.3.2 *Results.* All implementations except the looser tree show the same behavior: 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 the decrease in element movements, which is soon overtaken by the 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 of levels, and hence data movements, evolves as $1/\log(z)$. Unfortunately, the running time is twice as long as for the remaining implementations for $z = 2$, and barely catches up 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-way mergers. All these findings are rather consistent across the three architectures.

4.4 Merger Caching

In the outer recursion of Funnelsort, 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.

4.4.1 *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.

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

4.5 Base Sorting Algorithm

Like in any recursive algorithm, the base case in Lazy Funnelsort is handled specially. As a natural 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, for input sizes below αz^d we 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 sizes from 10 to 100 revealed the expected result, namely that `std::sort` (which in the GCC implementation itself switches to Insertionsort below size 16), 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 concern 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.

4.6.1 *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.

4.6.2 *Results.* There is a marked rise in running time when α drops 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 buffer sizes, 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 appear 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.

We implement two versions: `Funnelsort2`, which uses binary basic mergers as described in Section 2, and `Funnelsort4`, which uses the four-way basic mergers found in Section 4.3 to give slightly better results. The remaining implementation details follow what was declared the best choices in Section 4. Both implementations use parameters $(\alpha, d) = (16, 2)$, and use `std::sort` for input sizes below 400 (as this makes all k-mergers have height at least two in both implementations).

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 yardstick. Quicksort is known as a very fast general-purpose comparison-based algorithm [Sedgewick 1998], 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 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 v. 3.2 distribution, `std::sort` from the STL library from Dinkumware[5] included with Intels C++ compiler v.7.0, the implementation by Sedgewick from [Sedgewick 1998, Chap. 7], and an implementation of our own, based on a proposal from [Bentley and McIlroy 1993], but tuned slightly further by making it simpler for calls on small instances and adding an even 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. Two-way partitioning allows tighter code, but is less robust when repeated keys are present.

To gain further insight we also compare with recent 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 [Department of Computer Science, Duke University 2002] is a library for external memory computations, and includes highly optimized routines for e.g. scanning and sorting. We choose TPIE's 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 TPIE's manual we set it to 192 Mb, which is 50–75% of the physical memory on all machines where it is tested. The TPIE version available at time of experiments has release date August 29, 2002. It does not support the MIPS and Itanium architectures, and requires an older version (2.96) of the GCC compiler on the remaining architectures.

Several recent proposals for cache-aware sorting algorithms in internal memory exist, including [Arge et al. 2002; LaMarca and Ladner 1999; Xiao et al. 2000]. Proposals for better exploiting L1 and L2 cache were given in [LaMarca and Ladner

[5] `www.dinkumware.com`

1999]. Improving on their effort, [Arge et al. 2002] give proposals using registers better, and [Xiao et al. 2000] give variants of the algorithms from [LaMarca and Ladner 1999] 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 two Mergesort-based proposals from [LaMarca and Ladner 1999] as implemented by [Xiao et al. 2000] (we encountered problems with the remaining implementations from [Xiao et al. 2000]), and the R-merge algorithm of [Arge et al. 2002]. We use the publicly available code from [Xiao et al. 2000] and code from [Arge et al. 2002] sent to us by the authors.

5.2 Experiments

We test the algorithms described 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 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 less impact on these.

In these final experiments, besides the three machines used in Section 3, we also include the AMD Athlon and the Intel Itanium 2 processor.[6] Their specifications can be seen in Table I. The methodology is as described in Section 3.

5.3 Results

The plots described in this section are shown in Appendix A. In all graphs the y-axis shows wall time in seconds divided by $n \log n$, and the x-axis shows $\log n$, where n is the number of input elements.

The comparison of Quicksort implementations appear in Figures 4 and 5. Three contestants run pretty close, with the GCC implementation as the overall fastest. 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`), based on the tuned three-way partitioning of Bentley and McIlroy. The implementation from Sedgewick's book (denoted `Sedge`) follows closely after, whereas the implementation from the Dinkumware STL library (denoted `Dink`) lags rather behind, probably due to a rather involved three-way partitioning routine. We use the GCC and the Mix implementation as the Quicksort contestants in the remaining experiments—the first we choose for pure speed, the latter for having better robustness with almost no sacrifice in efficiency.

The main experiments in RAM appear in Figures 6 and 7. The Funnelsort algorithm with four-way basic mergers is consistently better than the one with binary basic mergers, except on the MIPS architecture, which has a very slow CPU. This indicates that the reduced number of element movements really do outweigh the increased merger complexity, except when CPU cycles are costly compared to memory accesses.

For the smallest input sizes the best Funnelsort looses to GCC Quicksort (by 10-40%), but on three architectures it gains as n grows, ending up winning (by

[6]We only had access to the Itanium machine for a limited period of time, and for this reason we do not have results for all algorithms on this architecture.

approximately the same ratio) for the largest instances in RAM. The two architectures where GCC keeps its lead are the MIPS 10000 with its slow CPU, and the Pentium 4, which features the PC800 bus (decreasing the access time to RAM), and which has a large cache line size (reducing effects of cache latency when scanning data in cache). This can be interpreted as follows: 1) on these two architectures, CPU cycles rather than cache effects dominate the running time for sorting, and 2) on architectures where this is not the case, the theoretically better cache performance of Funnelsort actually shows through in practice, at least for a tuned implementation of the algorithm.

The two cache-aware implementations `msort-c` and `msort-m` from [Xiao et al. 2000] are not competitive on any of the architectures. The R-merge algorithm is competing well, and like Funnelsort shows its cache-efficiency by having a basically horizontal graph throughout the entire RAM range on the architectures dominated by cache effects. However, four-way Funnelsort is slightly better than R-merge, except on the MIPS 10000 machine, where the opposite holds. This machine is a RISC-type architecture and has a large number of registers, something which the R-merge algorithm is designed to exploit. TPIE's algorithm is not competitive in RAM.

The experiments on disk appear in Figure 8, where TPIE is the clear winner. It is optimized for external memory, and we suspect in particular that the use of standard techniques for overlapping I/O and CPU activity (double-buffering and sorting one run while loading the next, techniques which seem hard to transfer to a cache-oblivious setting) gives it an unbeatable advantage. However, Funnelsort comes in as second, and outperforms GCC quite clearly. The gain over GCC seems to grow as n grows larger, which is in good correspondence with the difference in the base of logarithms in the I/O complexity of these algorithms. The algorithms tuned to cache perform notably badly on disk.

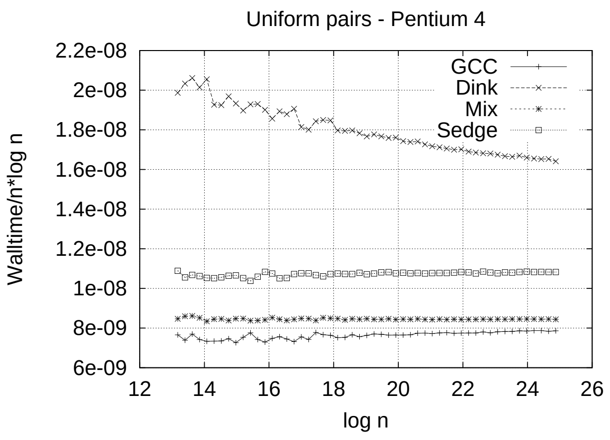
We have only shown plots for uniformly distributed data of the second data type (records of integer and pointer pairs). The results for the other types and distributions discussed in Section 3 are quite similar, and can be found in [Vinther 2003].

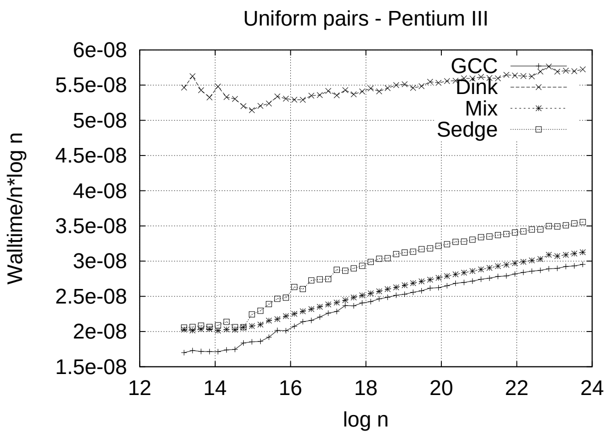
6. CONCLUSION

Through a careful engineering effort, we have developed a tuned implementation of Lazy Funnelsort, which we have compared empirically with efficient implementations of other comparison-based sorting algorithms. The results show that our implementation is competitive in RAM as well as on disk, in particular in situations where sorting is not CPU bound. Across the many input sizes tried, Funnelsort was almost always among the two fastest algorithms, and clearly the one adapting most gracefully to changes of level in the memory hierarchy.

In short, these results show that for sorting, the overhead involved in being cache-oblivious can be small enough for the nice theoretical properties to actually transfer into practical advantages.

A. PLOTS





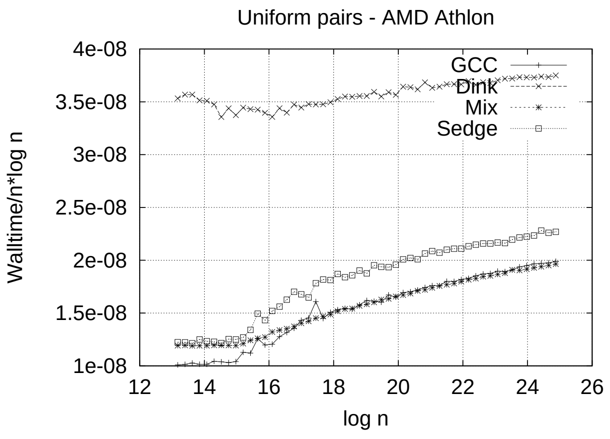
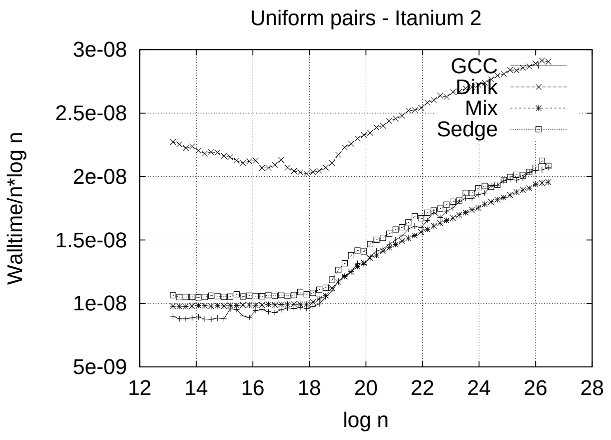


Fig. 4. Comparison of Quicksort implementations (part 1)



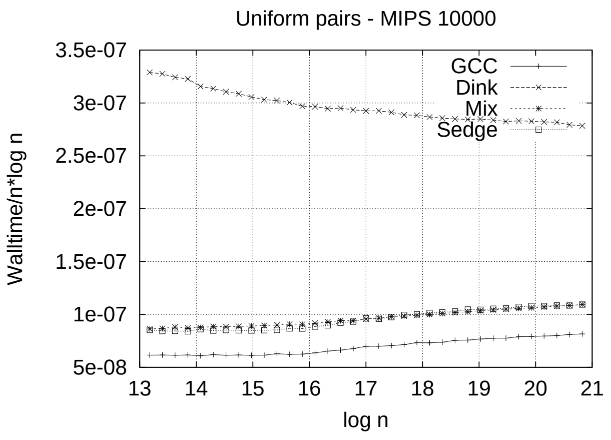
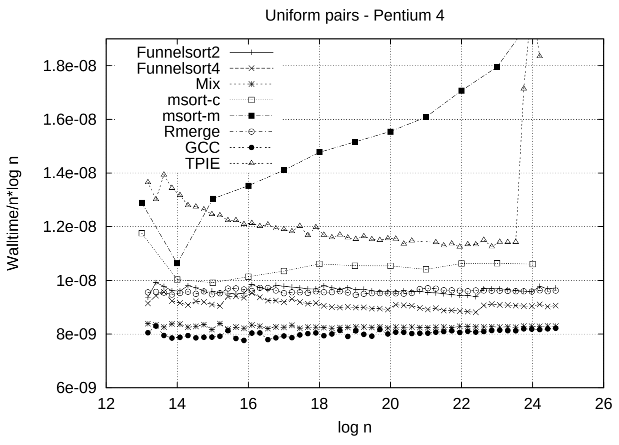
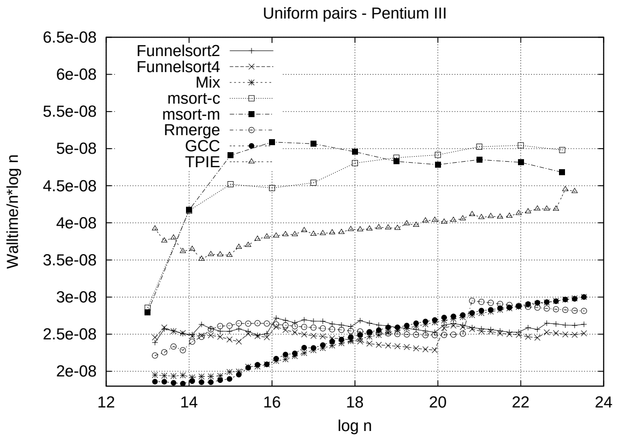


Fig. 5. Comparison of Quicksort implementations (part 2)





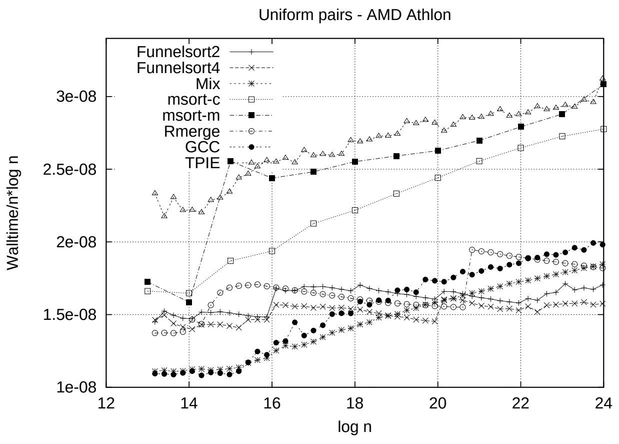
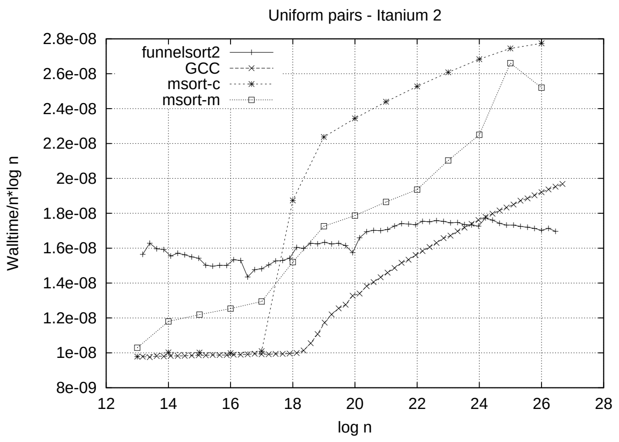


Fig. 6. Results for inputs in RAM (part 1)



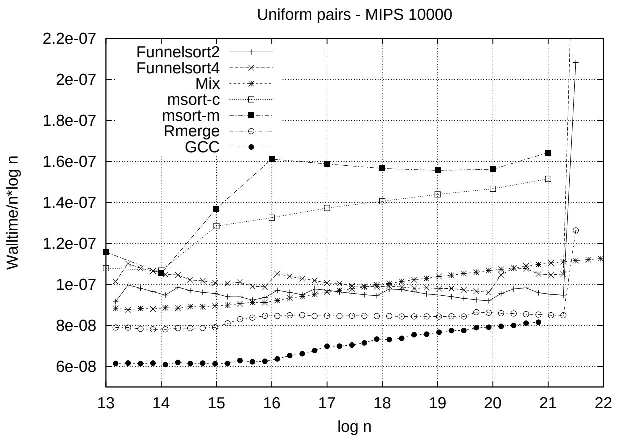


Fig. 7. Results for inputs in RAM (part 2)

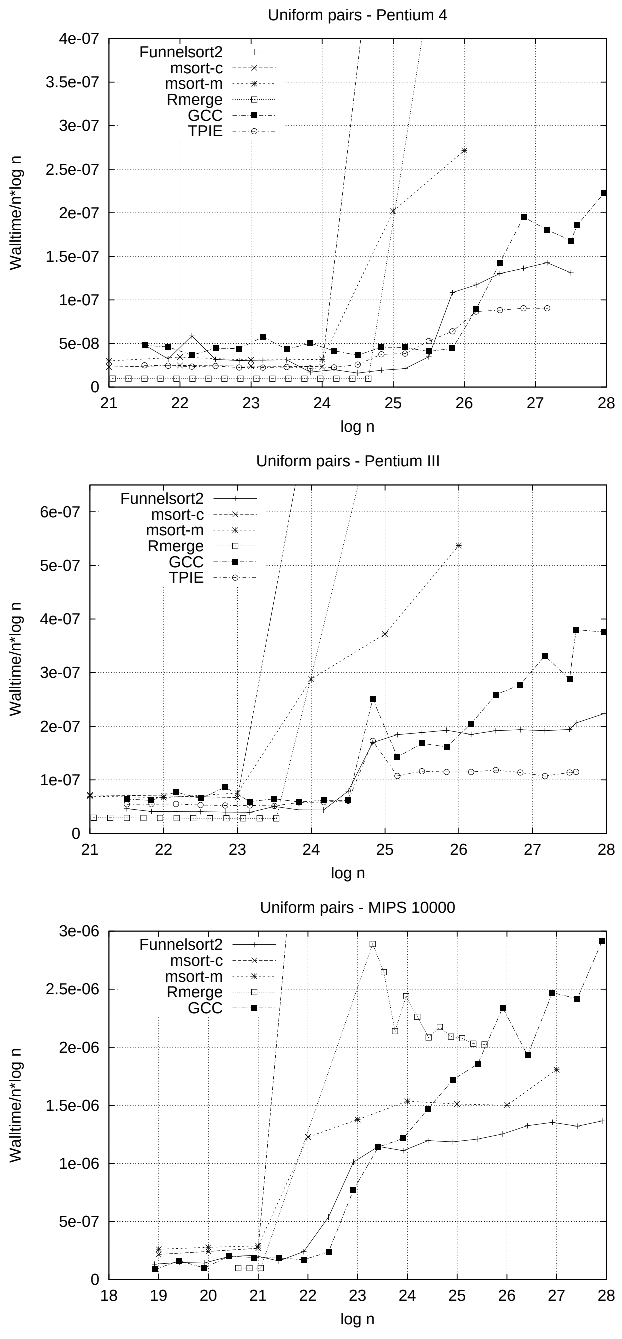


Fig. 8. Results for inputs on disk

ACKNOWLEDGMENT

We thank Brian Vinter of the Department of Mathematics and Computer Science, University of Southern Denmark, for access to an Itanium 2 processor.

REFERENCES

Agarwal, P. K., Arge, L., Danner, A., and Holland-Minkley, B. 2003. Cache-oblivious data structures for orthogonal range searching. In *Proc. 19th ACM Symposium on Computational Geometry.* ACM, 237–245.

Aggarwal, A. and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM 31,* 9, 1116–1127.

Arge, L. 2001. External memory data structures. In *Proc. 9th Annual European Symposium on Algorithms.* LNCS, vol. 2161. Springer, 1–29.

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

Arge, L., Brodal, G. S., and Fagerberg, R. 2005. Cache-oblivious data structures. In *Handbook of Data Structures and Applications*, D. Mehta and S. Sahni, Eds. CRC Press, Chapter 34.

Arge, L., Brodal, G. S., Fagerberg, R., and Laustsen, M. 2005. Cache-oblivious planar orthogonal range searching and counting. In *Proc. 21st Annual ACM Symposium on Computational Geometry.* ACM, 160–169.

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

Arge, L., de Berg, M., and Haverkort, H. J. 2005. Cache-oblivious R-trees. In *Proc. 21st Annual ACM Symposium on Computational Geometry.* ACM, 170–179.

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

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

Bender, M., Cole, R., and Raman, R. 2002. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming.* LNCS, vol. 2380. Springer, 195–207.

Bender, M., Demaine, E., and Farach-Colton, M. 2002. Efficient tree layout in a multilevel memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms.* LNCS, vol. 2461. Springer, 165–173. Full version at `http://arxiv.org/abs/cs/0211010`.

Bender, M. A., Brodal, G. S., Fagerberg, R., Ge, D., He, S., Hu, H., Iacono, J., and López-Ortiz, A. 2003. The cost of cache-oblivious searching. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science.* IEEE Computer Society Press, 271–282.

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

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

Bender, M. A., Farach-Colton, M., and Kuszmaul, B. C. 2006. Cache-oblivious string B-trees. In *Proc. 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM, 233–242.

Bender, M. A., Fineman, J. T., Gilbert, S., and Kuszmaul, B. C. 2005. Concurrent cache-oblivious B-trees. In *Proc. 17th Annual ACM Symposium on Parallel Algorithms.* ACM, 228–237.

Bentley, J. L. and McIlroy, M. D. 1993. Engineering a sort function. *Software–Practice and Experience 23,* 1, 1249–1265.

BRODAL, G. S. 2004. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*. LNCS, vol. 3111. Springer, 3–13.

BRODAL, G. S. AND FAGERBERG, R. 2002a. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*. LNCS, vol. 2380. Springer, 426–438.

BRODAL, G. S. AND FAGERBERG, R. 2002b. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*. LNCS, vol. 2518. Springer, 219–228.

BRODAL, G. S. AND FAGERBERG, R. 2003. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing*. ACM, 307–315.

BRODAL, G. S. AND FAGERBERG, R. 2006. Cache-oblivious string dictionaries. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 581–590.

BRODAL, G. S., FAGERBERG, R., AND JACOB, R. 2002. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 39–48.

BRODAL, G. S., FAGERBERG, R., MEYER, U., AND ZEH, N. 2004. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*. LNCS, vol. 3111. Springer, 480–492.

BRODAL, G. S., FAGERBERG, R., AND MORUZ, G. 2005. Cache-aware and cache-oblivious adaptive sorting. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming*. LNCS, vol. 3580. Springer, 576–588.

CHOWDHURY, R. A. AND RAMACHANDRAN, V. 2004. Cache-oblivious shortest paths in graphs using buffer heap. In *Proc. 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM.

CHOWDHURY, R. A. AND RAMACHANDRAN, V. 2006. Cache-oblivious dynamic programming. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 591–600.

DEPARTMENT OF COMPUTER SCIENCE, DUKE UNIVERSITY. 2002. TPIE: a transparent parallel I/O environment. WWW page, `http://www.cs.duke.edu/TPIE/`.

FAGERBERG, R., PAGH, A., AND PAGH, R. 2006. External string sorting: Faster and cache-oblivious. In *Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science*. LNCS, vol. 3884. Springer, 68–79.

FARZAN, A., FERRAGINA, P., FRANCESCHINI, G., AND MUNRO, J. I. 2005. Cache-oblivious comparison-based algorithms on multisets. In *Proc. 13th Annual European Symposium on Algorithms*. LNCS, vol. 3669. Springer, 305–316.

FRANCESCHINI, G. 2004. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 291–299.

FRANCESCHINI, G. AND GROSSI, R. 2003a. Optimal cache-oblivious implicit dictionaries. In *Proc. 30th International Colloquium on Automata, Languages, and Programming*. LNCS, vol. 2719. Springer, 316–331.

FRANCESCHINI, G. AND GROSSI, R. 2003b. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th International Workshop on Algorithms and Data Structures*. LNCS, vol. 2748. Springer, 114–126.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 285–297.

GRAY, J. 2003. Sort benchmark home page. WWW page, `http://research.microsoft.com/barc/SortBenchmark/`.

HWANG, F. K. AND LIN, S. 1972. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing 1,* 1, 31–39.

JAMPALA, H. AND ZEH, N. 2005. Cache-oblivious planar shortest paths. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming*. LNCS, vol. 3580. Springer, 563–575.

KNUTH, D. E. 1998. *The Art of Computer Programming, Vol 3, Sorting and Searching*, 2nd ed. Addison-Wesley, Reading, USA.

LADNER, R. E., FORTNA, R., AND NGUYEN, B.-H. 2002. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics.* LNCS, vol. 2547. Springer, 78–92.

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

PROKOP, H. 1999. Cache-oblivious algorithms. M.S. thesis, Massachusetts Institute of Technology.

RAHMAN, N., COLE, R., AND RAMAN, R. 2001. Optimised predecessor data structures for internal memory. In Proc. 5th International Workshop on Algorithm Engineering. *LNCS 2141*, 67–78.

SANDERS, P. 2000. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics 5,* 7.

SEDGEWICK, R. 1998. *Algorithms in C++: Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*, third ed. Addison-Wesley, Reading, MA, USA. Code available at `http://www.cs.princeton.edu/~rs/Algs3.cxx1-4/code.txt`.

VINTHER, K. 2003. Engineering cache-oblivious sorting algorithms. M.S. thesis, Department of Computer Science, University of Aarhus, Denmark. Available online at `http://kristoffer.vinther.name/academia/thesis/`.

VITTER, J. S. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys 33,* 2, 209–271.

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

XIAO, L., ZHANG, X., AND KUBRICHT, S. A. 2000. Improving memory performance of sorting algorithms. *ACM Journal of Experimental Algorithmics 5,* 3.

Received January 2005; revised August 2006; accepted October 2006