

***psort*, Yet Another Fast Stable Sorting Software**

PAOLO BERTASI, MARCO BRESSAN, and ENOCH PESERICO,

Università degli Studi di Padova

psort is the fastest sorting software according to the PennySort benchmark, sorting 181GB of data in 2008 and 224GB in 2009 for \$0.01 of computer time. This article details its internals, and the careful fitting of its architecture to the structure of modern PC-class platforms, allowing it to outperform state-of-the-art sorting software such as *STXXL sort*.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Design studies; E.5 [Files]: Sorting/searching; H.3.4 [Information Storage and Retrieval]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Sort, merge, stable, external, memory hierarchy, I/O, disk, RAID, Datamation, PennySort, database, C, algorithm engineering

ACM Reference Format:

Bertasi, P., Bressan, M., and Peserico, E. 2011. *psort*, Yet another fast stable sorting software. ACM J. Exp. Algor. 16, 2, Article 2.4 (June 2011), 19 pages.

DOI = 10.1145/1963190.1970377 <http://doi.acm.org/10.1145/1963190.1970377>

1. INTRODUCTION

This article details the internals of *psort*, the fastest sorting software of 2008 and of 2009 according to the PennySort benchmark [Sort Benchmark]. This introduction provides a brief history of PennySort and related sorting benchmarks (Section 1.1) and a simple taxonomy of the “mainstream” sorting techniques for large datasets that helps put our work into perspective (Section 1.2), followed by a high-level overview of *psort* and of the organization of the rest of this article (Section 1.3).

1.1. Datamation, PennySort, and Other Sorting Benchmarks

Datamation [Bitton et al. 1985] defined the first [Gray 2007] public sorting benchmark—sort a million 100-byte records, initially in random order, according to the first 10 bytes *from disk to disk*. In 1985 the time required to complete the Datamation benchmark was almost 1 hour; but within 10 years it had dropped to a few seconds, and it appeared that it would soon become a benchmark of a system’s *responsiveness* rather than throughput.

Supported in part by Univ. Padova under Strategic Project AACSE, by MIUR under Project AlgoDEEP, and by PAT, FBK, and INFN under Project Aurora-Science. Continued work on *psort* is supported in part by a generous equipment donation by Samsung. A shorter version of this article was presented at the 8th International Symposium on Experimental Algorithms [Bertasi et al. 2009].

Authors’ addresses: P. Bertasi, M. Bressan, and E. Peserico, Dip. Ingegneria dell’Informazione, Università degli Studi di Padova, Italy.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2011 ACM 1084-6654/2011/06-ART2.4 \$10.00

DOI 10.1145/1963190.1970377 <http://doi.acm.org/10.1145/1963190.1970377>

Therefore, Nyberg et al. [1995] proposed two new sorting benchmarks, MinuteSort and PennySort. MinuteSort, aimed at supercomputer-class platforms, requires sorting as many records as possible within 1 minute. PennySort, aimed at PC-class platforms, requires sorting as many records as possible with \$0.01 of computing time, assuming that the price of a machine is amortized over 3 years (thus, on a x dollar machine, one is allowed $\frac{0.01}{x} \cdot 3$ years of computing time). In both cases, record format coincides with that of the Datamation benchmark. Also, in both cases, a distinction is made between “Daytona” software, designed for general-purpose sorting, and “Indy” software, specifically optimized for the benchmark. Over the years a number of slight refinements have been added to the rules, and new benchmarks such as JouleSort have been introduced. All details can be found on the Sort Benchmark web site [Sort Benchmark].

1.2. A Simple Taxonomy of Sorting

Hundreds of articles and even entire books (e.g., Knuth [1998]) have been written on sorting. This section provides a simple taxonomy of sorting techniques for large datasets to help put *psort* into perspective compared to existing software.

Virtually all efficient sorting software today is either *distribution*-based, *merge*-based, or a hybrid of the two. Distribution-based sorting distributes the data into two or more bins, in such a way that for each pair of bins all keys of one precede all keys of the other; then it recursively sorts each bin. Merge-based sorting splits the input into two or more runs, sorts each run, and then merges the sorted runs. Distribution and merge-based sorting can obviously be combined. For example, one might use the former to “locally” sort separate runs that are in turn “globally” merged—the approach of AlphaSort [Nyberg et al. 1995].

Distribution-based sorting has two major advantages over merge-based sorting. First, it can be easily performed completely in parallel, and thus is virtually the only approach used today—at least at the “global” level—for sorting on large PC-clusters (e.g., Arpaci-Dusseau et al. [1997], DeWitt et al. [1991], Kuszmaul [2009], and Nodine and Vitter [1995]). Second, it can be very efficient in terms of number of key look-ups—this makes it a favorite of all past record holders of the PennySort benchmark (e.g., OzSort; Postman’s Sort [Postman’s Sort], Kuszmaul [2009], and Yang et al. [2003]) at least for some phases of the sort.

Merge-based sorting is instead intrinsically “comparison based” and thus requires at least $n \lg(n)$ key look-ups to sort n keys [Knuth 1998], though this disadvantage is more apparent than real, as we shall see. On the other hand, merge-based sorting always results in well-balanced subproblems of predictable size, which makes it more “robust” and easier to fine tune to a memory hierarchy. This predictability could in theory be achieved by a careful selection of the thresholds between bins in distribution-based sorting, but only at a high cost (see, e.g., Knuth [1998]).

1.3. Our Results

psort is a fast stable external sorting software (available as source, binary and/or library) that can sort collections of records of arbitrary size according to an arbitrary infix. *psort* sorted 181GB of data for \$0.01 of computer time in 2008 and 224GB in 2009, making it the fastest sorting software according to the PennySort benchmark. The careful fitting of its architecture to the structure of modern PC-class platforms (made easier by its pure merge-based nature) allows it to outperform widely used sorting software such as *GNU sort* [GNUSort], *qsort* [QSort], C++ STL *sort* [STLSort] or *STXXL sort* [Dementiev and Sanders 2003], even for record and key sizes and distributions quite different from those of the PennySort benchmark.

In order to understand *psort* optimizations, one has to understand the complex architecture of modern PCs to a greater level of detail than that offered by most theoretical

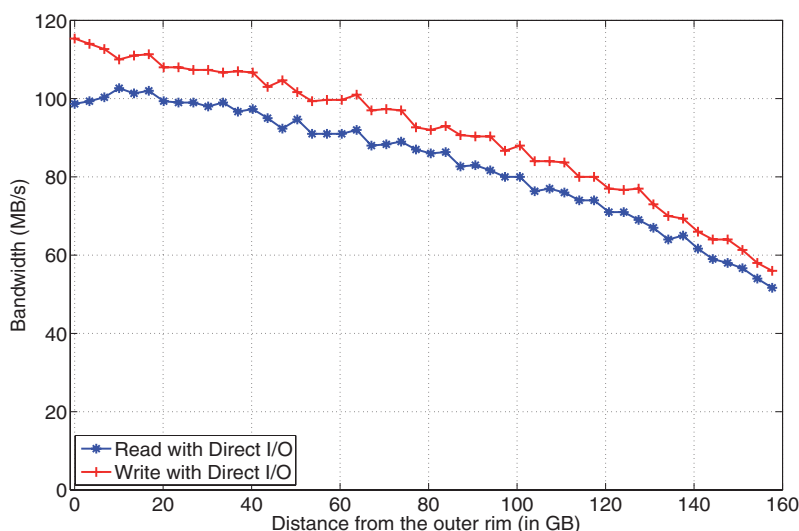


Fig. 1. Disk bandwidth (for data transfers of 64MB) as a function of the distance from the outer rim, on a Western Digital WD1600AAJS-00B4A0 160GB disk.

models today. Section 2 provides this information. Section 3 describes *psort* itself and its tuning to a modern PC. Section 4 describes our experimental results, comparing *psort* with other state of the art sorting software, both under the PennySort rules (including the selection of the “best” PC to run it) and under a number of different scenarios. Finally, Section 5 summarizes our results and discusses their significance.

2. THE ANATOMY OF A PC

Current hardware is extremely complex. While a number of abstract models from the past decades attempt to capture the main aspects of modern architectures—hierarchical memory [Aggarwal et al. 1987], block transfer [Aggarwal et al. 1987; Alpern et al. 1994], pipelining [Bilardi et al. 2002], parallelism [Valiant 1990] etc.—they are generally insufficient to abandon the ivory tower of big O notation and to squeeze out of a machine at least 50% of its peak performance [Ailamaki et al. 1999]. It is impossible to review all the details of the performance of a modern PC, but this section provides a comprehensive overview of those (often disregarded) factors that can be crucial to data-intensive software such as sorting, starting from disks and file system (Section 2.1), then moving to motherboard and memory (Section 2.2), and finally to the processor chip (Section 2.3).

2.1. Disks and File System

Modern disks provide the abstraction of a (logical) linear array of data blocks, with access to a sequence of contiguous blocks requiring a fixed *seek time* independent of the amount of data (typically of the order of 10ms), plus a *transfer time* that is directly proportional to it (typically 10ms/MB to 100ms/MB). Obviously, to approach peak performance, transfer time should dominate seek time (contiguous data transfers indicatively of 1MB or more). Transfer time is lower for data logically closer to the beginning of the array, corresponding physically to the area of the disk closer to the outer rim; this should be regarded as the “high-performance” portion of the drive (see Figure 1).

Multiple disks (usually up to four to eight) can be used in parallel (RAID 0) as a single disk with the same seek time but proportionally larger transfer speed, by splitting data into “stripes” divided, in a round-robin fashion, between different disks. This can be

done in hardware or by the file system/OS; we found the latter approach effective and of minimal CPU cost.

Most software today does not access the disk directly, but through a file system. File systems offer a lot of functionality over raw disk access, but at a price in terms of performance (in fact, many applications with high-performance disk access, such as DBMSs, bypass the file system altogether); however, most sorting benchmarks (including PennySort) enforce the use of a file system for disk access. To minimize the CPU and memory system overhead, it is then crucial to access data through asynchronous, direct I/O from/to the device directly into/from user space.

2.2. Motherboard and Main Memory

Disks (or rather their on-board caches) communicate with the motherboard through an interface generally designed to have higher throughput (e.g., 300MB/s for SATA2) than the drives themselves, to ensure interface longevity. Motherboards typically support four to six (rarely eight) of these interfaces, all connected to a chip known as the *south bridge*. The south bridge aggregates the traffic of disks and other devices (e.g., keyboard, network card) and usually directs it to another chip, the *north bridge*, that controls traffic to/from/between the main memory, the processor chip and graphics hardware (in some architectures, at least part of the north bridge is incorporated in the processor).

Modern south and north bridges can manage transfers of large data aggregates directly between memory and disk with only minimal CPU involvement (Direct Memory Access). Bandwidth between bridges is typically lower than the sum of the disk interface bandwidths supported by the motherboard, but is rarely a bottleneck for all but the fastest and largest same-generation disk RAIDs (typical values are 350MB/s to 800MB/s). The bandwidth between the other hardware connected to the north bridge is higher still, typically several GB/s.

There are a few more issues to consider in terms of processor/memory communication. First, over a hundred processor cycles can typically pass between a request for a datum in main memory and its availability on the processor chip. Second, data are transferred to the processor chip only in multiples of relatively large *cache lines* (typically 64 to 512 bytes); and if 2 bytes of a datum belong to 2 different cache lines, both lines are accessed. Sophisticated line-prefetch mechanisms are sometimes used to accelerate large transfers of contiguous data. Third, virtual memory addresses have to be translated to physical addresses. While the most recent translations are cached in the processor chip, very sparse memory accesses will force additional accesses to the translation tables [Rahman et al. 2001]. This problem can be minimized using large memory pages, and thus smaller translation tables for the same space (see Figure 2). The bottom line is that the assumption of memory being truly “random access” is unrealistic, even if it is read or written in blocks of size equal to a cache line; generally speaking, large, sequential reads and writes tend to exhibit better performance (see Figure 3).

2.3. The Processor Chip

The basic components of the processor chip are its CPU(s) and its cache(s). Most modern processors have two levels of cache, the first smaller (up to a few hundred KB vs. one or more MB), but faster to access (typically a few cycles vs. 10–30). Each memory block can only be placed in a small number of cache locations, the *associativity* of the cache (typically 4 to 16 for L1 caches, 8 to 32 for L2 caches). According to Hennessy and Patterson [1996] an 8-way associative cache generally provides the same performance as a fully associative one, but we have found this to be true only if data layout is carefully planned as in Mehlhorn and Sanders [2003]. All decisions on which data to keep in the cache are made by the processor, typically replacing data used furthest

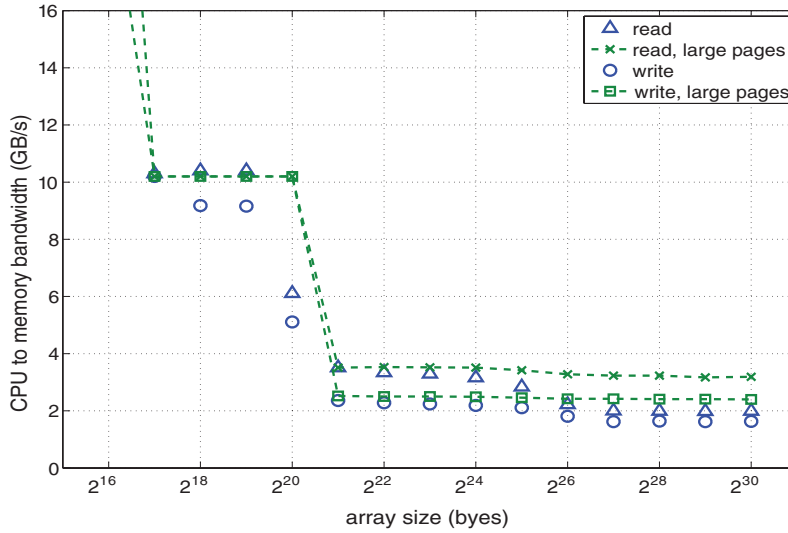


Fig. 2. Memory bandwidth when sequentially accessing arrays of different sizes, positioned randomly in memory, using standard pages (4KB) and “huge” pages (2MB) on an Athlon LE1640 system with PC6400 DDR2 RAM and a Gigabyte GA-MA74GM-S2h motherboard. The two points where the curves diverge correspond to the overflow of the TLB information from, respectively, the L1 and the L2 cache.

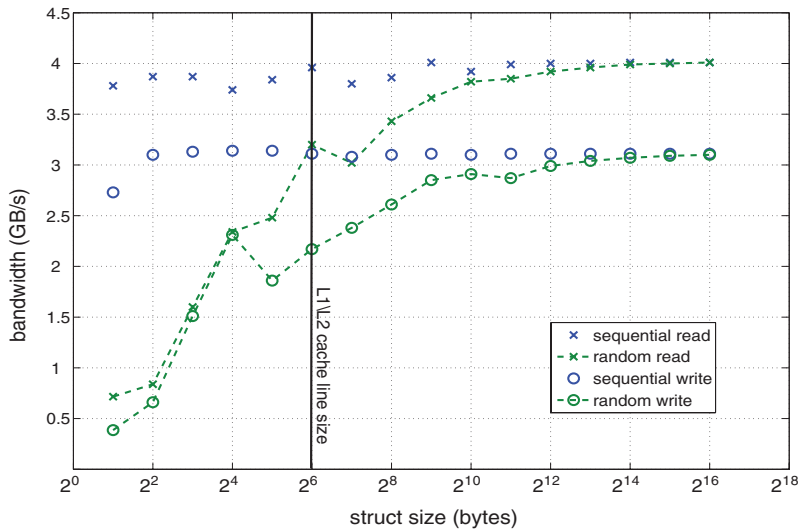


Fig. 3. Memory bandwidth when sequentially accessing arrays of different sizes, positioned in memory either randomly or sequentially (effectively a single large array), on an Athlon LE1640 system with PC6400 DDR2 RAM and a Gigabyte GA-MA74GM-S2h motherboard. Line size was 64 bytes for both the L1 and the L2 cache.

in the past with data more recently accessed. This can result in undesired behaviors; for example, large streams of read-once data can “pollute” the cache, evicting data used repeatedly but somewhat infrequently. The programmer can sometimes attempt to influence cache replacement by issuing extra memory requests to “nail” or prefetch data into the cache.

In general, modern processors employ sophisticated circuitry to reorganize instructions, “guess” yet unavailable data, and backtrack from incorrect guesses. This makes it extremely difficult to understand whether, and how, a code snippet can be redesigned to improve efficiency: “optimizations” that increase performance on one system often decrease it on others. Fine tuning through experimentation on the target architecture can net substantial performance gains.

3. THE ANATOMY OF *PSORT*

psort is a fast, stable sorting software designed for large datasets on PC-class platforms. *psort* is a simple, merge-based sorter that first sorts individual data runs approximately the size of main memory, and then merges them into a single sorted output. In fact, *psort*'s high-level simplicity is probably the source of its performance, allowing careful, low-level tuning to the complex structure of today's PCs. Section 3.1 provides a “global overview” of *psort*. Sections 3.2 and 3.3 provide the details of the first and second phase of the sort.

3.1. A Theorist's View of *psort*

psort is a merge-based sorter tuned to the memory hierarchy. For each pair of consecutive hierarchy layers (e.g., main memory and disk) the structure of *psort* depends on the size S_0 of the smaller and on the size S_1 of the larger.

In a nutshell, *psort* implements a p -pass merge between the two layers as follows. Choose a block size B such that $(\frac{S_0}{B})^p \approx \frac{S_1}{B}$. To implement the first pass, split the data in the larger layer into $\approx \frac{S_1}{S_0}$ data runs of size $\approx S_0$, and sort each run in the smaller layer. Data are sorted in the smallest layer, the L1 cache, using a simple mergesort. To implement the second pass, split the sorted runs into sets of $w = \frac{S_0}{B}$ runs each, and merge each set into a sorted run of size wS_0 using a w -way selection tree merger [Knuth 1998], where each way is assigned a buffer of size B . The entire tree then always resides in cache, so that during the pass each item is brought into, and evicted from, the cache only once, and each transfer is amortized over a block of size B . Similarly, to implement the i^{th} pass, merge sets of w runs from the $(i-1)^{\text{th}}$ pass into sorted runs of size $w^{i-1}S_0$.

It is important to note that each block should contain data that are stored *contiguously* on the medium in order to avoid fragmentation and guarantee efficient transfers (see Section 2.1). The minimum block size that still allows an efficient transfer places an upper bound on the number of ways of a merge, and thus a lower bound on the number p of passes. More precisely, let t_B be the amortized read time per bit using blocks of size B , and t_{peak} be the amortized write time per bit at the peak transfer rate using blocks of size comparable to S_0 (during a merge phase, the input buffers must be small, but the single write buffer can be large). Then B is chosen to minimize the amortized total merge time per bit, which is $(p-1)(t_B + t_{\text{peak}})$. Note that the first term of the product is $(p-1)$: in the first pass, one can essentially perform both reads and writes of a size comparable to S_0 . In practice, on modern computing platforms, p almost always equals 2 or (more rarely) 3 at all layers of the memory hierarchy, with the possible exception of the lowest two (memory and disk), where datasets small enough to fit in memory can be sorted with a single read/write pass of the disk(s). This is not the case for the PennySort benchmark, where data is usually one to two orders of magnitude larger than memory—making $p = 2$ and thus entailing 2 “large writes” passes, 1 “large

reads” pass (the first), and 1 “small reads” pass (the second). In this case B is chosen to match the size of a few disk tracks (i.e., a few MB—to minimize the overhead of seek time) times the number of disks in the RAID.

Two things are worth noting. First, the first pass may always be performed in place, with at most a “slack” of a single run; this is generally impossible for other passes without seriously compromising performance. Second, as noted, for example, in Dementiev and Sanders [2003], utilizing RAID at the disk layer is asymptotically suboptimal (as the number of disks grows to infinity) compared to (randomized) strategies that control disks independently. However, their “sweet spot” requires larger disk arrays and sorts than those encountered in practice when dealing with today’s PC-class platforms, where they can actually end up being slower even on “average” inputs due to their small (in fact *asymptotically* negligible) fluctuations in disk utilization (see Section 4.3) and their reduced ability to exploit the faster zones of the disk (see Section 4.2).

3.2. The First Phase

The first phase of *psort* essentially involves reading a data run approximately the size of the main memory from disk, sorting it in memory, and writing it back to disk. The devil is in the details.

For I/O efficiency, *psort* makes use of direct asynchronous I/O that transfers data between the disks and a set of user space buffers (dimensioned so as to achieve near peak transfer rate without consuming too much memory): this requires minimal CPU involvement (even with software RAID, less than 4% of CPU time on the architectures we tested) and bypasses the space and time costs of moving data through kernel space buffers. With two buffers one can guarantee that the disks never fall idle: while the CPU operates (reading or writing data) on one, disks exchange data with the other, thus completely overlapping I/O with computation.

As soon as a read buffer is filled, *psort* must transfer data to its main memory space. This transfer causes the data to transit through the processor’s cache; *psort* then exploits the temporary availability of the data in the cache to carry out a portion of the computation at this time, avoiding a read and a write access to the memory later on. More precisely, *psort* operates on datablocks slightly smaller than the L2 cache, which we call *microruns*, as follows. Each microrun is read from the input buffer. Then, if keys are sufficiently small compared to the records’ remaining payload, *psort* separates keys and payloads, attaching to each key a pointer to the corresponding payload. At this stage, *psort* also offers the possibility of restructuring keys (e.g., from big to small endian) to make later comparisons more efficient. *psort* then sorts those keys (and eventually reshuffles the payloads) and finally writes the whole “microrun” to a new memory area. This involves, for each datum, at most one write and one read in memory that would be performed anyway to empty the read buffers.

The sorting algorithm used in L1 cache for the microruns is a simple mergesort, with some tweaks. First, a single pass on the data (the same that possibly detaches keys and records) sorts sets of up to 8 consecutive elements using a simple selection sort with all loops unrolled; this appears to run slightly faster (5% to 7%) than using a similarly optimized insertion sort on random and worst-case inputs. The choice of set size offers an interesting trade-off between code size and code speed (see Figure 4).

Second, careful placement of data ensures that only 5/4ths of the total space s occupied by the records (or by the detached keys) is used, rather than the “common” factor 2 of mergesort. This is achieved by first sorting half of a microrun in space $\frac{s}{2} + \frac{1}{2} \cdot \frac{s}{2}$ (see Knuth [1998]), then the other half (for a total space of $\frac{s}{2} + \frac{s}{2} + \frac{1}{2} \cdot \frac{s}{2} = \frac{5s}{4}$), and finally piggybacking the merge of the two halves over the transfer out of the read buffer. This can net significant performance gains when microrun size is close to (L1 or L2) cache size.

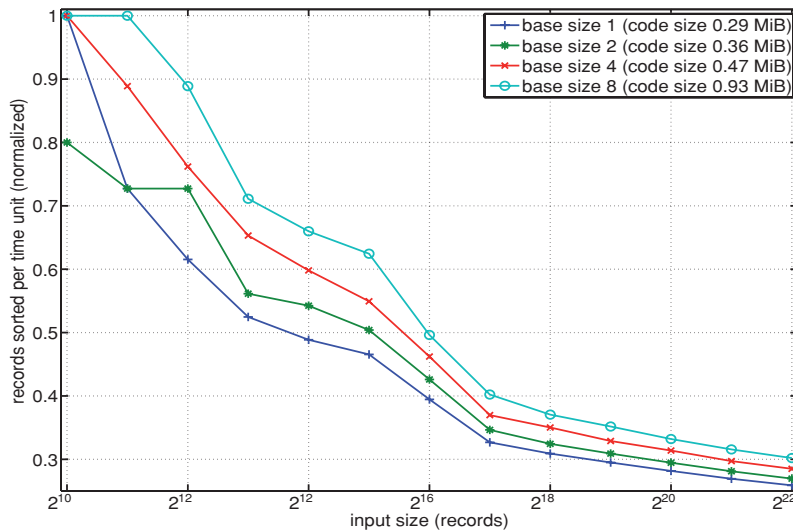


Fig. 4. Relative speed of the in-cache sorter for different base set sizes, as a function of the input size, on an Athlon LE1640 system with PC6400 DDR2 RAM and a Gigabyte GA-MA74GM-S2h motherboard.

Third, *psort* offers the opportunity to minimize program branches. As a first option, it can use bitwise—rather than logical—ANDs and ORs when comparing multiword keys. As a second option, it can use the result of a comparison (and its inverse) as an offset to a pointer to the new positions of the keys, saving a branch at the cost of a few extra operations. We have found that the effects of these two optimizations are hard to predict a priori.

The data sorted in L1 are merged in L2. Sorted microruns are then merged in a second pass, and possibly a third, depending on the ratio between the size of the memory and that of the L2 cache, and the associativity of the latter.

Two important potential hurdles at this stage are “stream pollution” of the cache (see Section 2) and associativity misses (in early experiments, these reduced performance by as much as 20%). Against the former, *psort* offers the option of a periodic cache refresh through dummy reads. To minimize the latter, *psort* employs a careful data layout similar to that of Mehlhorn and Sanders [2003]: microruns are placed in such a way that the addresses of their initial elements are evenly distributed over the cache, rather than being all mapped to the same cache line(s).

The output of the final pass is directly written to the I/O buffers, potentially recombining keys and payloads, and possibly (if there is no second phase) inverting the initial restructuring of the keys. Again, double buffering allows full overlap between I/O and computation.

3.3. The Second Phase

The second phase (which only takes place if the dataset does not fit in main memory) is much simpler: w sorted runs at a time are streamed from disk and merged (with the same code that merges microruns in the first phase), and the output is streamed back to disk. Recalling Section 3.1, it is reasonable to use a single merge pass if the number of data runs (i.e., the ratio between data and memory size) does not exceed the ratio between the size of the memory and that of one “efficient” read from disk.

Data is read with direct asynchronous I/O into (user space) dynamically sized buffers, one per run. When the amount of data in a buffer falls below a threshold, the buffer is

“refilled” from the appropriate run. In theory, if data were consumed uniformly from all w runs, one could divide the total available buffer space B in such a way that a newly refilled buffer held $\approx \frac{w}{w^2+w}2B \approx 2B/w$ bytes of data, the previously refilled one $\frac{w-1}{w^2+w}2B$, and so on. This would allow reads of about twice the size achievable with static buffers of size B/w . This can be highly ineffective, however, if data are not consumed uniformly, and in particular if they are consumed more rapidly from recently refilled buffers. For this reason, *psort* allows the user to specify buffer geometry, choosing a trade-off between average and worst case performance.

4. PSORT VS. THE COMPETITION

This section compares *psort* to its competitors, in terms of the PennySort benchmark (Sections 4.1 and 4.2) and also in a wider variety of sorting scenarios (Section 4.3).

4.1. Choosing and Configuring the Hardware

To test *psort* we looked for a hardware platform delivering maximum performance at the minimum cost, to maximize our results under the PennySort benchmark, but also to understand the bottlenecks in today’s PC architectures for data-intensive tasks.

In 2008 our motherboard choice was an ASRock ALive NF6P-VSTA with an Nvidia nForce 430 Southbridge, an inexpensive but high-performance “Linux friendly” motherboard supporting four SATA2 channels, for a maximum aggregated traffic of 500MB/s. We paired it with four Western Digital WD1600AAJS drives, which can individually deliver a whopping 110MB/s peak read/write rate.

Somewhat surprisingly, we noticed a considerable variance in performance between different disks (see Figures 5 and 6). This variability was apparently not due to external factors (e.g., being connected to different ports, being placed in a slightly warmer region of the case, etc.); even reshuffling the disks, each maintained its “signature” performance curve.

We configured the disks with GNU/Linux (Gentoo) “vanilla” software RAID. As a file system, we tested SGI’s XFS, IBM’s JFS, ReiserFS, and ext2fs. The best performers were XFS and JFS, with JFS slightly better overall but XFS outperforming it very slightly for the read and write sizes of interest to us (see Figures 7 and 8). In both cases, CPU usage to saturate the disk transfer rate was negligible—less than 2%. Thus, we finally settled for XFS. The best performance was achieved with a stripe size of 128KB. Note that this is a very “disk-heavy” PC, with about half the total cost being taken by the four disks (see Figure 9).

In 2009, our motherboard choice changed to that of a Gigabyte GA-MA74GM-S2, which supported six SATA2 channels, allowing us to increase the number of disks to five. In fact, performance was even higher using six disks, but not enough to offset the increased cost. We used the same disks as in 2008: due to budget limitations, we could not afford to test more than a few extra models, and none outperformed our Western Digital WD1600AAJS in terms of performance per dollar. In hindsight, more extensive tests would have yielded a significant advantage: *OzSort* [OzSort], which almost tied with *psort* in the Indy category [Sort Benchmark], employed five disks with a price/peak bandwidth ratio almost 12.7% lower.

RAM choice must take into account three parameters: size, speed, and price. A RAM that is twice as large more than doubles the size of runs in the first pass. This, in turn, allows reads that are over four times longer during the second pass. It turned out that the best compromise was 2GB in 2008, and 4GB in 2009. RAM speed is another important parameter. PC 4200RAM has a *theoretical* “peak” transfer rate of 4.2GB/s—an order of magnitude faster than the southern bridge. In practice, we found that accessing RAM can have a large number of “hidden” costs (e.g., due TLB look-ups

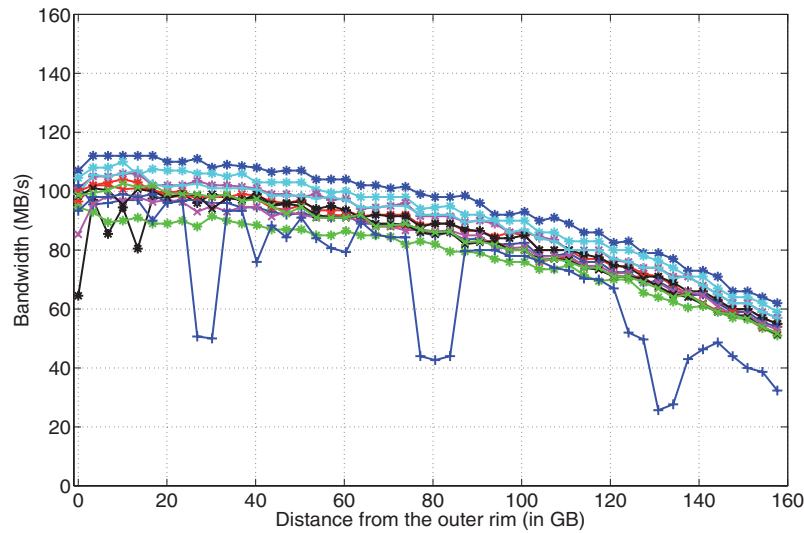


Fig. 5. Read speed of 13 Western Digital WD1600AAJS drives as a function of the distance from the outer rim of the disk.

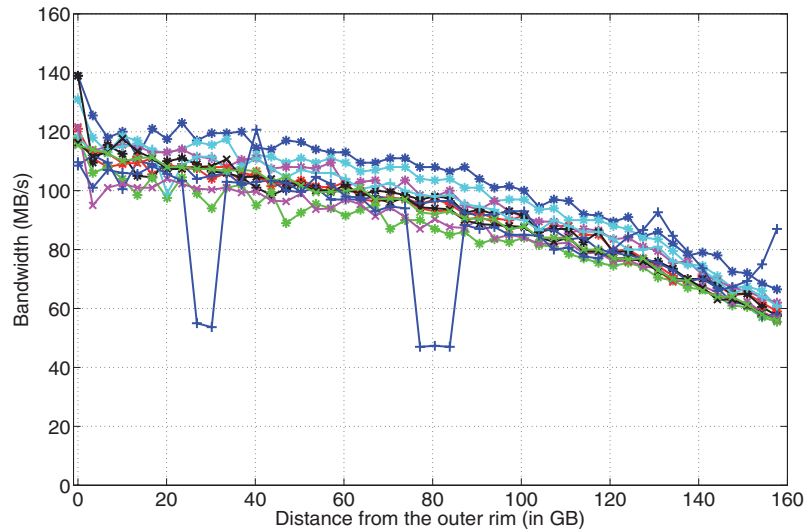


Fig. 6. Write speed of 13 Western Digital WD1600AAJS drives as a function of the distance from the outer rim of the disk.

and to the fact that each access involves the transfer of a full cache line). Even just two or three read-and-write passes can consume the majority of the available RAM bandwidth, and it is extremely difficult to coax the compiler to overlap RAM to cache transfers with processor operations. In practice, it turned out that even using two banks of OCZ 800MHz PC2-6400 Ram with CAS 4 latency almost half the “CPU” time during the first pass was spent accessing RAM. In 2009 we switched to PC2-6400 Ram with CAS 5 latency which appeared to yield comparable performance for a considerably better price.

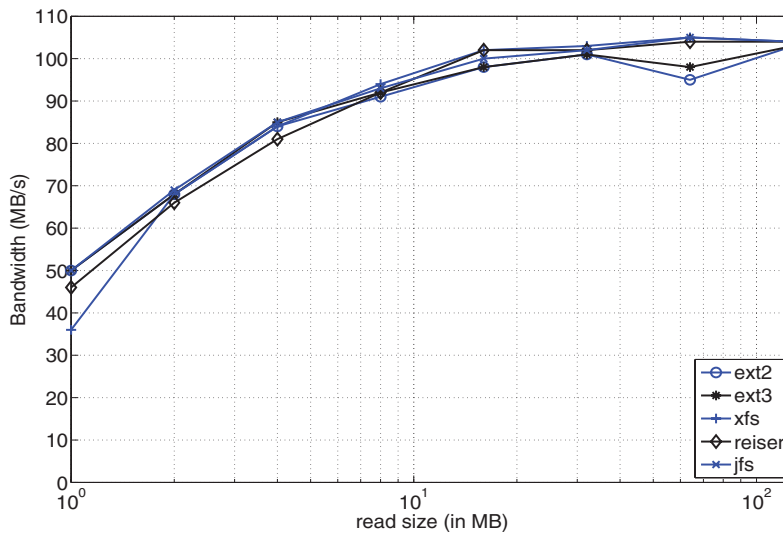


Fig. 7. File system read speed as a function of read size.

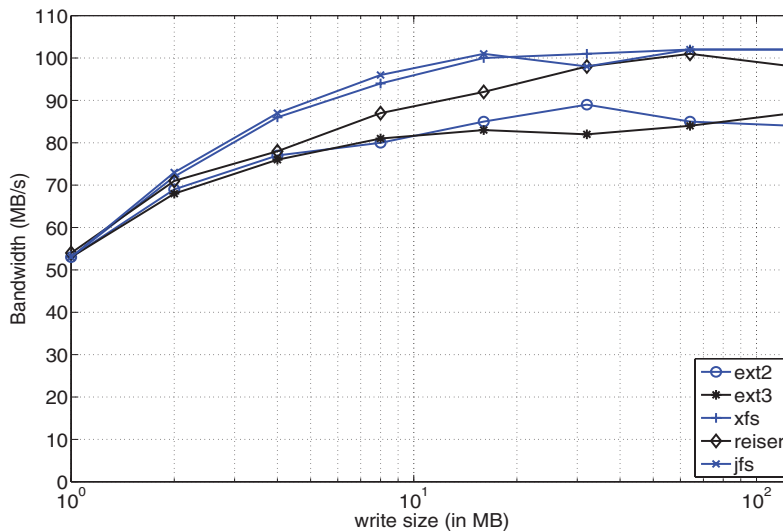


Fig. 8. File system write speed as a function of write size.

The choice of the actual processor strongly depends on that of the other components, probably more than on the “number crunching” power of the CPU itself. In 2008 we chose a cheap, single-core Athlon 1620LE running at 2.4GHz, with 128KB of L1 cache and 1MB of L2 cache. It would have yielded adequate performance in 2009 as well, but unfortunately the model was no longer available in major online stores (an important constraint of the PennySort benchmark is that all components should be purchasable at the same store). Thus, we opted for a slightly faster clocked, but otherwise identical, Athlon 1640LE running at 2.6GHz.

The total cost of the hardware at NewEgg.com on May 19, 2008 was \$357.78 (see Figure 9). Under the PennySort formula, adding the mandatory \$35 “assembly fee,”

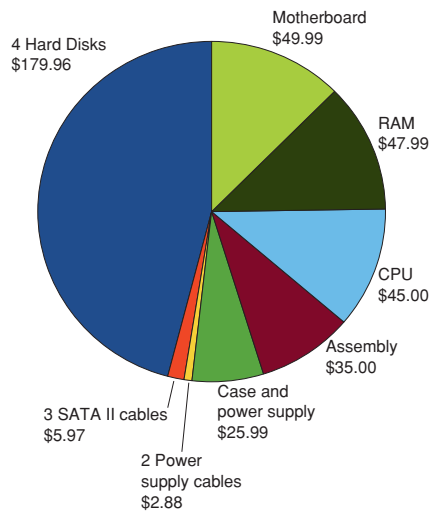


Fig. 9. Budget of the testbed machine (2008).

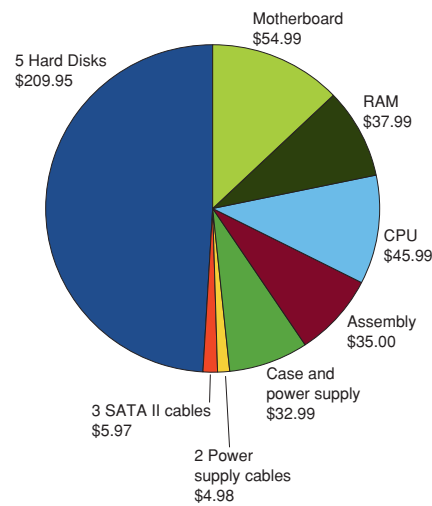


Fig. 10. Budget of the testbed machine (2009).

this allowed us a total time budget of slightly more than 2,408.67 seconds. In 2009 the total cost of the hardware was \$427.86, (including the assembly fee—see Figure 10), yielding a slightly smaller time budget of 2,211.19 seconds.

4.2. PennySort Results

In 2008 we tested *psort* on the hardware described in Section 4.1. We compiled using gcc version 4.1.2 with the flags:

```
-march=k8 -O3 -funroll-loops -funsafe-loop-optimizations
-B /usr/share/libhugetlbfs/ -Wl,--hugetlbfs-link=B.
```

We positioned the input file into an appropriately sized partition on the outer rim of the disks, overwrote it with the output of the first pass, and had the second pass create the output file in a second partition. This guaranteed three of the four passes took place on the fastest partition of the disk, and only one on a slower partition. Note that had we been using independent disks for the intermediate files (RAID would still have been necessary for the initial input and final output, since the rules of the PennySort benchmark enforce a single file for each), at most two passes could have taken place on the faster partition.

The first pass was slightly limited by the CPU or, more correctly, by the combination of CPU and RAM. More expensive CPUs did not yield sufficient increases in performance to justify their use. The second pass was entirely limited by the disks. *psort* (using 2^{16} -record microruns and a 2^8 -way selection tree, 50MB read/write buffers, overwriting the initial file with the intermediate file) sorted $108 \cdot 2^{24} = 1,811,939,328$ records taking less than 2,405 seconds. We then manually retooled *psort* into an “Indy” version adapted solely for the PennySort benchmark (eliminating unnecessary “general-purpose sorting” code, manually unrolling loops, etc.). This yielded a small, but observable gain in performance. *psort* Indy managed to sort $113 \cdot 2^{24} = 1,895,825,408$ records in less than 2,407 seconds.

In 2009, the rules for the Daytona benchmark changed, and one was no longer allowed to overwrite the input file. This forced us to use *three* separate partitions. We placed the input into the innermost (slowest) partition, wrote the output of the first pass

into the outermost (fastest) partition, and the final output into the middle partition. This allowed us to use the fastest partition twice and to use the slowest partition (once) during the first phase, when the CPU-memory subsystem is a slightly tighter bottleneck and reads and writes are sequential. We used the same version of gcc (4.1.2) and the same flags of 2008. Within the time budget of 2,210 seconds, *psort* managed to sort $67 \cdot 2^{25} = 224,8146,944$ records. The hand optimized Indy version (using the same data layout as in 2008) managed to sort $74 \cdot 2^{25} = 2,483,027,968$ records, slightly outperforming *OzSort* (by approximately 2GB—the difference was sufficiently small that a 2-way tie was declared).

4.3. Other Scenarios

In order to evaluate *psort* outside the PennySort context, we compared it to some state-of-the-art sorting software under different scenarios that cover both external sorting (Section 4.3.1) and in-memory sorting (Section 4.3.2). We denote by (key_size, record_size) an input consisting of record_size-byte records to be sorted according to the first key_size bytes.

4.3.1. External Sorting: nsort and STXXL. We had *psort* compete against Ordinal’s external sorting software *nsort* [Nyberg et al. 1997] and the high-performance (non-stable) external sorting *STXXL* library [Dementiev and Sanders 2003]. Tests were performed according to the numerical and lexicographical order of keys on (8, 8), (8, 128), (128, 128) inputs, excluding only the numerical ordering of (128, 128) inputs for *nsort* (which cannot handle numerical keys of more than 8 bytes [Nsort]). The size of the input ranged from 1GB to 128GB, thus we only ran *STXXL sort* with key sizes that were multiples of a word size, in line with Dementiev and Sanders [2003].

We ran *nsort* using radix as sort type and pointer as sort kind, except for the input (8, 8) where record was used. The tuning was left to *nsort* itself, as suggested in the manual. Numerical order and lexicographical order were obtained specifying respectively unsigned bin and character as key kind. To improve the performance, direct I/O was used via the direct flag of the -out_file option. Due to the portability limitations of the trial version, *nsort* was run on a 64-bit CentOS 5.5 instead of the Gentoo we used for all the other experiments.

We ran *STXXL sort* modifying the algo/test_sort.cpp test source code. A record was represented by a struct containing an array of integers (the key) and an array of chars (the payload). The comparison operators were implemented via math operators (for numerical order) or calling the strcmp() function (for lexicographical order). To improve the performance, direct I/O was used via the stxxl::file::DIRECT flag of the stxxl::syscall_file() function. For the container, we used the stxxl::vector template using the best arguments we found. We used stxxl::lru_pager as pager type and striping as allocation strategy type; the optimal number of blocks per page and optimal pages in cache varied from 1 to 4 and the optimal block size appeared to be 256MB.

Figure 11 shows numerical ordering results, and Figure 12 shows lexicographical ordering results. In comparison to both *nsort* and *STXXL sort*, *psort* was generally slightly faster, and significantly faster in the case of large records. Note that *psort* implements an optional feature to increase the performance when the input contains large “well-distributed” keys: it first performs an “approximated” sorting pass according to a short prefix of the keys and then, according to the whole keys, sorts all the records that are still unordered. This option was used with an 8-byte prefix on (128, 128) inputs and explains the good performance of *psort*.

It should be noted that *STXXL sort*, like *psort*, has a large number of tuning parameters that we extensively tweaked trying to achieve optimal performance; we would

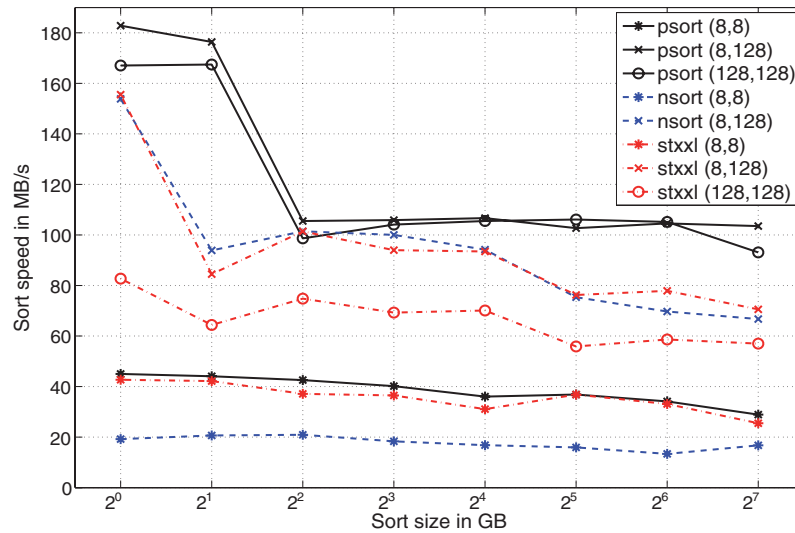


Fig. 11. *psort* vs. *nsort* and *STXXL sort*, sorting records according to the numerical ordering of the keys, for various combinations of (key_size, record_size).

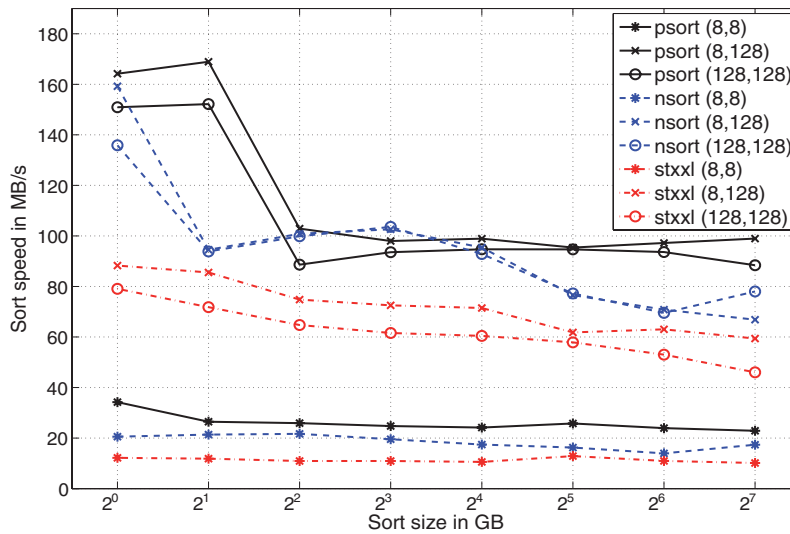


Fig. 12. *psort* vs. *nsort*, and *STXXL sort*, sorting records according to the lexicographical ordering of the keys, for various combinations of (key_size, record_size).

thank Andreas Beckmann, one of the two maintainers of *STXXL*, for his insightful and continued *STXXL* optimization support. Interestingly, and contrary to what one might expect from the article by Dementiev and Sanders [2003], using independent disks never increased performance significantly, and without extensive parameter tweaking in many cases it even slightly decreased it, due to the fluctuations it introduces in disk usage. Apparently the theoretical advantage of using independent disks translates into a significant practical advantage only for significantly larger sorts.

4.3.2. In-Memory Sorting: *qsort*, GNU *sort*, and STL *sort*. We also had *psort* compete in main memory against *qsort* [QSort], GNU *sort* [GNUSort] and the C++ STL library [STLSort] (which do not support external memory sorting) on sorting (8, 8), (8, 32) and (8, 128) inputs according to the numerical order of the keys, as well as (10, 100) inputs according to the lexicographical order of the keys (the Datamation record format), again for a variety of input sizes ranging from few megabytes to several hundred megabytes.

Input and output files were placed in a ram disk. Note that this put *psort* at a disadvantage compared to in-memory sorting software, since *psort* is designed to overlap computation and disk accesses: transferring data from and to a ram disk causes *psort*'s "disk" accesses to interfere with its memory accesses. Still, *psort* was always 30% to 400% faster than *qsort* and GNU *sort*, neither of which is stable, with a wider gap for large record sizes (see Figures 13 and 14). We tested the STL routines `stable_sort()` and `sort()`; *psort* was always 10% to 400% faster than the first, and was significantly outperformed by the second only on (8, 8) inputs (see Figures 15 and 16).

We also tested a modified version of STL `sort()` that incorporated two major optimizations of *psort*. More precisely, keys were first detached and extended with a progressive id, then sorted with `sort()`, and finally reattached to their respective payloads. Note that this made the resulting sort stable even if the underlying `sort()` routine is non-stable. The second optimization involved restructuring the keys in the case of lexicographical ordering. For record sizes larger than key sizes this yielded a significant speed-up, up to a factor 3 or 4, over `stable_sort()` (see Figures 15 and 16). This highlights the importance of these optimizations. Still, *psort* performed better on every input.

Note that *STXXL* uses STL routines for in-memory sorting, yet STL is outperformed by *STXXL*, which may sound counterintuitive. This is likely due to the careful implementation of data types (such as containers) and algorithms (such as iterators) in *STXXL*, resulting in lower CPU/memory usage than in "vanilla" STL `sort()` (see Demetiev and Sanders [2003] for more details).

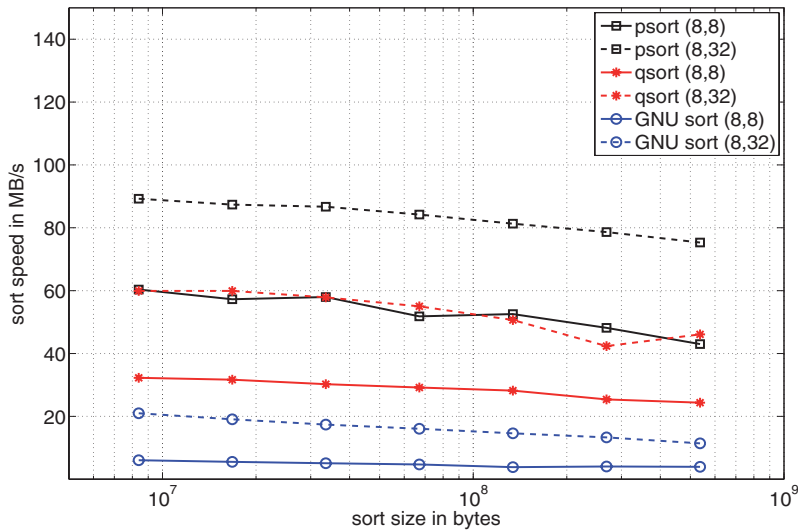
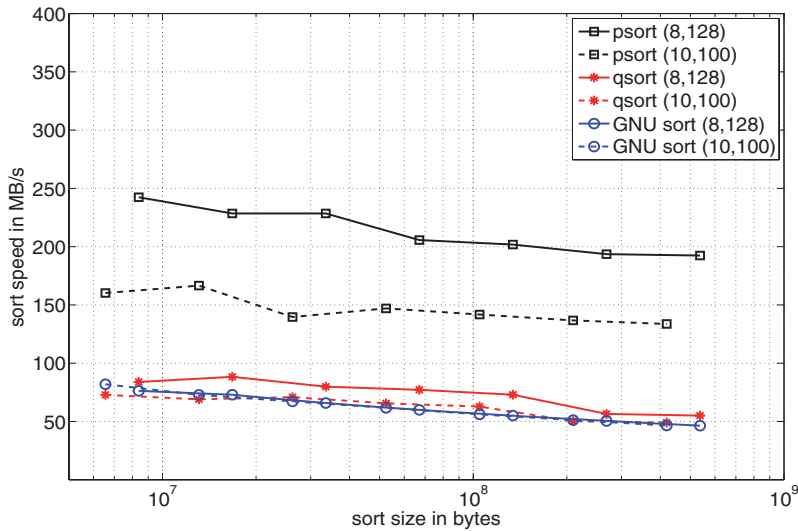
5. CONCLUSIONS

This section briefly summarizes our results, discusses their significance, and looks at future directions both in terms of efficient sorting and other data-intensive software (Section 5.2) and of the PennySort benchmark (Section 5.1).

5.1. A Decade of PennySort

It is interesting to compare our results with the prediction of 1998 by Gray et al. [1998] that price-performance would double yearly for the next 10 years, yielding 1.50TB for one penny by 2008. Instead, price-performance has "only" increased by an average factor of about 1.6/year (see Figure 17). This almost exactly matches Moore's Law; but, looking at the 1998 winners, it is easy to see that improvement is not due solely to better hardware. In 2008 we sorted more than 120 times the data of Gray et al. [1998] using 3 times the time budget, a set of disks with about 15 times the peak (total) bandwidth, and a 2.4GHz Athlon 1620LE vs. a 266MHz Pentium II—the latter having a lesser comparative gap to the memory. Software engineering advances are then responsible for at least a factor 2 to 3 of improvement (note that *psort*'s basic algorithms are decades old).

After more than 10 years, PennySort is still an excellent benchmark for the lower levels of the memory hierarchy—less so for the processor. As reflected by our budget (see Figures 9 and 10), our machines had superb disks, excellent motherboards, good memory, and two of the cheapest processors on the market. While the PennySort "spirit" has remained the same over the years, the rules keep changing slightly every year. We

Fig. 13. *psort* vs. *qsort* and *GNU sort* for small record sizes.Fig. 14. *psort* vs. *qsort* and *GNU sort* for large record sizes.

believe this reduces the value of PennySort as a benchmark for the *evolution* of PCs and sorting software. On the other hand, we advocate one change: stipulating that the prices used to compute the time budget be taken from a list made public a few months before the submission deadline. This would avoid last minute shopping (and coding) frenzies and/or heavy impact on the relative results of different entries caused by fluctuations of hardware prices.

5.2. Some (Ugly?) Lessons from *psort*

Unlike all previous winners of the PennySort benchmark, *psort* is completely merge-based, rather than a distribution-based hybrid. This might be one reason of its success.

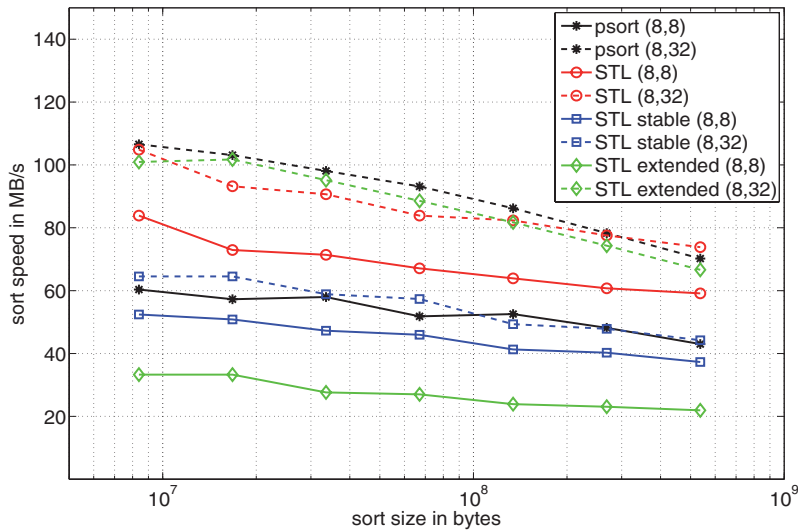


Fig. 15. *psort* vs. *STL sort*, *STL stable.sort*, and *STL sort (extended keys)* for small record sizes.

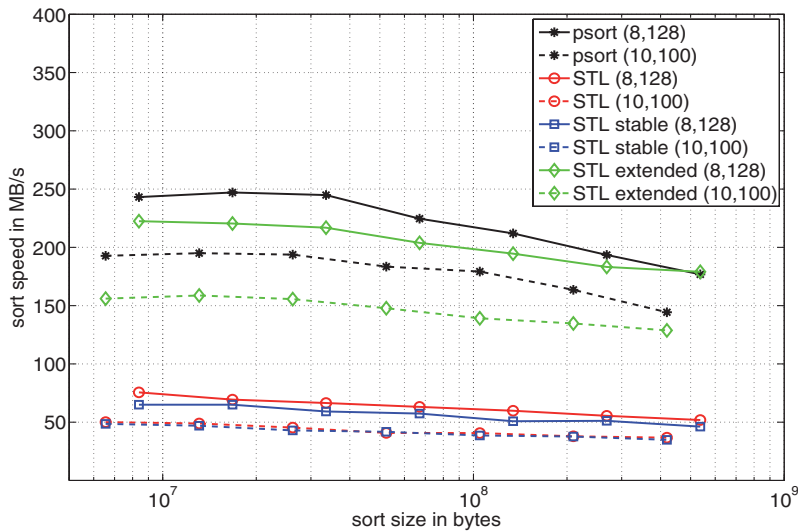


Fig. 16. *psort* vs. *STL sort*, *STL stable.sort*, and *STL sort (extended keys)* for large record sizes.

Merge-based software tends to require more key look-ups at the highest levels of the memory hierarchy, but these levels are no longer the bottleneck. On the other hand, merge-based sorting software is somewhat more predictable—and thus it can be fitted more carefully to the lower levels of the memory hierarchy, avoiding performance losses where they count.

psort exploits many simple tricks that can be expected to boost the performance of any data-intensive software. Perhaps the ultimate lesson of *psort* is that a lot of ugly work is necessary to transform any simple, elegant algorithm into a software that preserves at least 50% of the performance potential of today's PCs. This explains why sorting, despite its relative simplicity, its long history, and its great practical importance, can

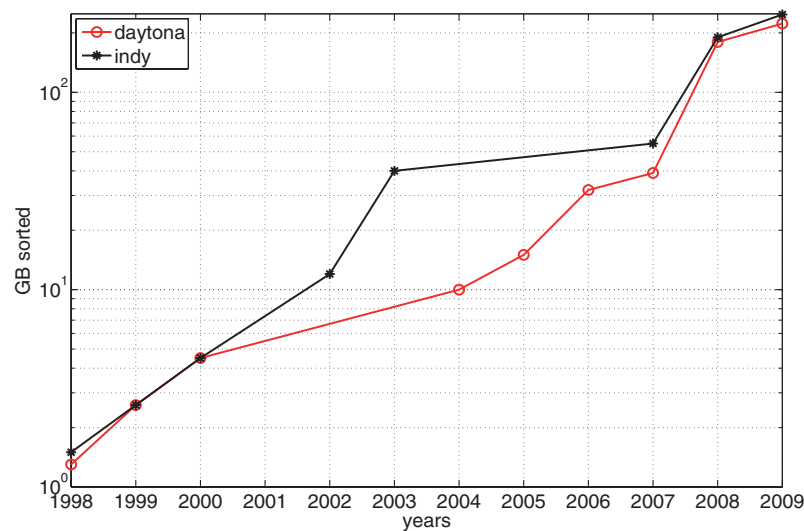


Fig. 17. The PennySort Benchmark record history (*psort* = 2008 and 2009).

still see nontrivial improvements through simple algorithm engineering (rather than algorithmic breakthroughs).

REFERENCES

- AGGARWAL, A., ALPERN, B., CHANDRA, A., AND SNIR, M. 1987. A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC'87)*. ACM, New York, 305–314.
- AGGARWAL, A., CHANDRA, A., AND SNIR, M. 1987. Hierarchical memory with block transfer. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science (FOCS'87)*. IEEE, Los Alamitos, CA, 204–216.
- AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. DBMSs on a modern processor: where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*. Springer, Berlin, 266–277.
- ALPERN, B., CARTER, L., FEIG, E., AND SELKER, T. 1994. The uniform memory hierarchy model of computation. *Algorithmica* 12, 2/3, 72–109.
- ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. 1997. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 243–254.
- BERTASI, P., BRESSAN, M., AND PESERICO, E. 2009. *psort*, yet another fast stable sorting software. In *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*. Springer, Berlin, 76–88.
- BILARDI, G., EKANADHAM, K., AND PATTNAIK, P. 2002. Optimal organizations for pipelined hierarchical memories. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms (SPAA'02)*. ACM, New York, 109–116.
- BITTON, D., BROWN, M., CATELL, R., CERI, S., CHOU, T., DEWITT, D., GAWLICK, D., GARCIA-MOLINA, H., GOOD, B., ET AL. 1985. A measure of transaction processing power. *Datamation* 31, 7, 112–118.
- DEMENTIEV, R. AND SANDERS, P. 2003. Asynchronous parallel disk sorting. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms (SPAA'03)*. ACM, New York, 138–148.
- DEWITT, D. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1991. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS'91)*. IEEE, Los Alamitos, CA, 280–291.
- GNUSort. GNU Coreutils, sort. <http://www.gnu.org/software/coreutils/>.
- GRAY, J. 2007. A measure of transaction processing 20 years later. *CoRR abs/cs/0701162*.
- GRAY, J., COATES, J., AND NYBERG, C. 1998. Performance/price sort. *CoRR cs.DB/9809004*.
- HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*, 2nd Ed. Kaufmann, San Francisco.

- KNUTH, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed. Addison-Wesley.
- KUSZMAUL, B. C. 2009. Brief announcement: TeraByte TokuSampleSort sorts 1TB in 197s. In *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms (SPAA'09)*. ACM, New York, 127–129.
- MEHLHORN, K. AND SANDERS, P. 2003. Scanning multiple sequences via cache memory. *Algorithmica* 35, 1, 75–93.
- NODINE, M. H. AND VITTER, J. S. 1995. Greed sort: optimal deterministic sorting on parallel disks. *J. ACM* 42, 4, 919–933.
- Nsort. Ordinal Technology. Nsort User Guide. <http://www.ordinal.com/NsortUserGuide.pdf>.
- NYBERG, C., BARCLAY, T., CVETANOVIC, Z., GRAY, J., AND LOMET, D. B. 1995. AlphaSort: a cache-sensitive parallel external sort. *Int. J. Very Large Databases (VLDB Journal)* 4, 4, 603–627.
- NYBERG, C., KOESTER, C., AND GRAY, J. 1997. Nsort: a parallel sorting program for NUMA and SMP machines. Ordinal Technologies white paper, 1997. <http://www.ordinal.com/white/whitepaper.htm>.
- OzSort. <http://www.ozsort.com/>.
- QSort. GNU C library, Array Sort Function. http://www.gnu.org/s/libc/manual/html_node/Array-Sort-Function.html.
- RAHMAN, N., COLE, R., AND RAMAN, R. 2001. Optimised predecessor data structures for internal memory. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*. Springer, Berlin, 67–78.
- POSTMAN'S SORT. http://www.rrsd.com/software.development/postmans_sort/.
- Sort Benchmark. Home page. <http://sortbenchmark.org/>.
- STLSort. GNU C++ library, std namespace. <http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/namespacestd.html>.
- VALIANT, L. G. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 103–111.
- YANG, L., HUANG, H., WAN, Z., AND SONG, T. 2003. SheenkSort: 2003 Performance/Price Sort and PennySort. <http://sortbenchmark.org/SheenkSort.pdf>.

Received November 2009; revised March 2011; accepted March 2011