

# Engineering a Cache-Oblivious Sorting Algorithm

Brodal, Fagerberg, and Vinther (ACM JEA, 2007)

---

Presented by Abrianna Zhang

Sep 29, 2026

# Outline

Motivation

Background and Definitions

Related Work

Engineering Study

Discussion

# Motivation

---

# Why should sorting care about memory?

- Modern machines have a **memory hierarchy**: registers  $\rightarrow$  L1  $\rightarrow$  L2  $\rightarrow$  L3  $\rightarrow$  RAM  $\rightarrow$  disk.
- Access time grows sharply at each level, especially from RAM to disk.
- The cost of an algorithm depends heavily on where in the hierarchy the data it touches currently lives.
- Classic RAM model of analysis (instant read/write) is blind to this — two algorithms with same asymptotic operation count can have very different running times irl.



# The (two-level) I/O model

## Aggarwal & Vitter, 1988

- Fast memory of size  $M$ , slow memory (disk); data moves in blocks of  $B$  elements.
- Cost of a computation = number of block transfers.

Comparison-based sorting of  $N$  elements requires  $\Theta(\text{Sort}_{M,B}(N))$  I/Os, where

$$\text{Sort}_{M,B}(N) = \frac{N}{B} \log_{M/B} \frac{N}{B}.$$

## Catch

The algorithm has to know  $M$  and  $B$  to hit this bound — not portable across a multi-level hierarchy, and not robust if available memory changes (e.g. other processes running).

# Cache-oblivious algorithms

**Frigo, Leiserson, Prokop, Ramachandran, 1999**

An algorithm is **cache-oblivious** if it is written with no reference to  $M$  or  $B$ , yet its I/O cost can still be analyzed in the I/O model, for arbitrary  $M, B$ .

- Assumes an idealized, optimal offline cache replacement strategy.
- Since the bound holds for every  $M, B$ , it holds *simultaneously* at every level of a real multi-level hierarchy.
- Big payoffs: portability (same code, no tuning per machine) and robustness (memory pressure from other processes doesn't break the guarantee).

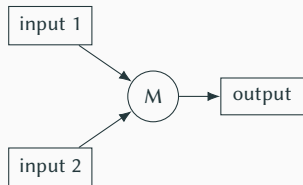
This paper examines whether such theoretical elegance transfers to real machines.

## **Background and Definitions**

---

# Binary mergers and merge trees

- A **binary merger** merges two sorted input streams into one sorted output stream, one element at a time.
- Heads of the inputs and the tail of the output live in small **buffers**.
- Chain mergers together: output of one = input of the next  $\Rightarrow$  a **binary merge tree**.
- Leaves hold the raw streams to merge.

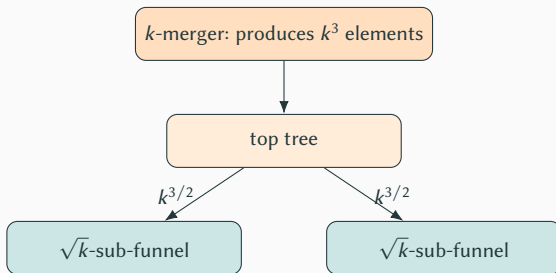


## **FILL( $v$ ) – invoking a merger**

**while** output buffer of  $v$  is not full:    **if** an input buffer is empty, recursively FILL that child first; then perform one merge step.

# The $k$ -merger

- A  $k$ -merger:  $k - 1$  binary mergers forming a complete binary tree of height  $\lceil \log k \rceil$ , with  $k$  input streams at the leaves and an output buffer of size  $k^d$  at the root ( $d > 1$  a design parameter).
- Buffer sizes are defined *recursively* — the key structural idea.



$\sqrt{k}$  buffers of size  $k^{3/2}$  each  $\Rightarrow$  total intermediate storage =  $\sqrt{k} \cdot k^{3/2} = k^2$ .

## The algorithm

To sort  $N$  elements: recursively sort  $N^{1/d}$  segments of size  $N^{1-1/d}$ , then merge the sorted segments with a single  $N^{1/d}$ -merger.

- “Lazy”: an input buffer is refilled only when it runs empty.
- Needs a **tall-cache assumption**  $M = \Omega(B^2)$ .
- Five cache-oblivious sorting algorithms existed at the time; Lazy Funnelsort is the *structurally simplest*  $\Rightarrow$  most promising to implement well.

# A simplifying observation (this paper's own contribution)

## Prior belief

Earlier descriptions of the  $k$ -merger require a specific recursive memory layout (van Emde Boas layout) for the I/O bound to hold.

## What the authors show

Re-examining the original proof: what actually matters is the *sizes* of the buffers, not how the merger is laid out in memory. Any layout gives the same asymptotic bound.

- Frees the implementation to choose whatever layout is fastest in practice (Section 4).
- Caveat: the whole  $k$ -merger should still occupy one contiguous block of memory for the complexity proof of Funnelsort itself to go through.

## Related Work

---

# Where this sits in the literature

**Cache-oblivious sorting** (5 known algorithms, same asymptotic bound):

- Funnelsort (Frigo et al. 1999)
- **Lazy Funnelsort** (Brodal–Fagerberg) – simplest, used here
- Distribution-based sort (Frigo et al.)
- Implicit sort (Franceschini 2004)
- Adaptive sort (Brodal et al. 2005)

Prior *empirical* work on cache-obliviousness covered matrix algorithms, DP, and search trees — this paper extends that tradition to sorting.

**Broader cache-oblivious model:**

- Search trees, B-trees
- Priority queues
- Comp. geometry, graphs, DP
- Tall-cache is *necessary* for optimal comparison sorting (Brodal–Fagerberg 2003)

# Engineering Study

---

**Machines** (tuning phase + 2 more for final test):

- Pentium 4 (modern CISC)
- Pentium III (classic CISC)
- MIPS 10000 (RISC)
- + AMD Athlon (modern CISC)
- + Itanium 2 (EPIC)

**Setup:**

- C++, max compiler optimization
- 3 element types: int; int+pointer record; 100-byte record
- Mainly uniform random keys; wall-clock time (`gettimeofday`)
- Parameters tuned *one at a time*, in a fixed order

# Design space for the $k$ -merger

## Variations of $k$ -Merger structure

- Tree layout: BFS / DFS / vEB
- Node + buffer storage: together / separate
- Navigation: pointers / index arithmetic
- Invocation: recursive / iterative
- Allocation: custom / default

28 combinations  $\times$  3 machines.

Slowest combo up to 65% slower; biggest factor was recursive vs. iterative invocation (esp. on Pentium 4). vEB layout gave  $\sim 10\%$  — spatial locality matters in practice, even though it's invisible to the asymptotic analysis.

### Best combination

|            |           |
|------------|-----------|
| Invocation | recursive |
| Navigation | pointers  |
| Layout     | vEB       |
| Storage    | separate  |
| Allocation | default   |

## Tuning the inner merge loop

- **Min-count trick:** compute  $\min(\text{elements left}, \text{space left})$  once, then merge that many steps without re-checking buffer state (removes a branch).
- **Hwang-Lin/Knuth optimal merge:** asymptotically better when input sizes are very skewed, but has higher constant overhead; also tried a hybrid that only invokes it on skewed buffers.

### Result: the plain, straightforward loop wins

The min-count heuristic was actually *15–45% slower*; Hwang-Lin alone was  $3\times$  slower. Modern branch predictors already handle the (mostly predictable) branches in the simple loop as well as any hand-coded trick — so the “cleverness” just adds overhead.

## Degree of the basic mergers

- A  $k$ -way merger doesn't need to be a binary merger — a  $z$ -way basic merger removes every other tree level: less data movement and navigation, but more comparisons per merge step.
- 7 implementations tested for  $z = 2, \dots, 9$ .

### Result

Time first drops, then rises with  $z$ ; minimum around  $z = 4-5$  ( $\sim 10\%$  faster than binary).

# Merger caching & choosing the base case

## Merger caching

- Reuse the same-size  $k$ -merger across all calls at a given recursion level instead of rebuilding it.
- **Result: 3–5% speedup**, consistent across machines.

## Base case algorithm

- Below a size threshold, use classic sorts instead of  $k$ -merger.
- Tried Insertion/Selection/Heap/Shell sort and STL `std::sort`.
- **Result: `std::sort` wins** at every tested size — used as the base case.

## Tuning $\alpha$ and $d$

- $d$ : controls the outer recursion split and the buffer sizes;  $\alpha$ : an extra multiplicative constant on buffer sizes, introduced by the authors themselves for tuning.
- Swept  $d \in [1.5, 3]$ ,  $\alpha \in [1, 40]$  on RAM-sized inputs.

### Result

Running time rises sharply once  $\alpha < 10$  (up to  $4\times$  slower at  $\alpha = 1$ ), worst at  $d = 1.5$ : smaller buffers mean the fixed cost of invoking a merger is amortized over fewer merge steps.

Chosen:  $\alpha \approx 16$ ,  $d \approx 2.5$ .

**Two final implementations** (all tuned choices,  $(\alpha, d) = (16, 2)$ , base case below size 400):

- **Funnelsort2** – binary basic mergers
- **Funnelsort4** – four-way basic mergers

## Competitors:

- **Quicksort**: GCC `std::sort`, Dinkumware `std::sort`, Sedgewick's implementation, authors' own implementation (mostly changing partitioning)
- **Cache-aware**: TPIE (external-memory library, double-buffered I/O), `msort-c/msort-m` (LaMarca–Ladner, via Xiao et al.), R-merge (Arge et al., register-tuned).

## Results in RAM

- Funnelsort4 usually beats Funnelsort2 (less data movement) — except on slow-CPU MIPS, where extra per-step work isn't worth it.
- Funnelsort loses to GCC Quicksort on small inputs (10–40% slower), but overtakes it as  $N$  grows — *except* on two machines.
- `msort-c/msort-m` aren't competitive anywhere; TPIE isn't competitive in RAM (it's built for disk).

| Architecture | Winner for large $N$ , in RAM              |
|--------------|--------------------------------------------|
| Pentium 4    | GCC Quicksort (fast bus, wide cache lines) |
| Pentium III  | Funnelsort                                 |
| MIPS 10000   | GCC Quicksort (slow CPU dominates)         |
| AMD Athlon   | Funnelsort                                 |
| Itanium 2    | Funnelsort                                 |

- **TPIE wins clearly** — it's purpose-built for external memory and overlaps I/O with computation (double buffering), a technique the authors note is hard to transfer to a cache-oblivious setting.
- **Funnelsort is a clear second**, and beats GCC Quicksort by a growing margin as  $N$  grows — consistent with Funnelsort's optimal log base vs. Quicksort's  $O(\frac{N}{B} \log_2 \frac{N}{M})$ .
- The *cache-aware* algorithms (msort, R-merge), tuned for RAM caches, perform notably badly once data spills to disk.

## Paper's conclusions

- A carefully engineered Lazy Funnelsort is competitive with, and often faster than, the best Quicksort and cache-aware implementations available — both in RAM and on disk (TPIE on disk being the one clear exception).
- Across the whole range of input sizes tested, Funnelsort was almost always among the two fastest algorithms, and adapted most gracefully across different levels of the memory hierarchy.
- Takeaway: for sorting, the theoretical overhead of being cache-*oblivious* (vs. cache-*aware*) can be small enough that the theory's benefits show up in practice.

## Discussion

---

- Very thorough: systematic tuning tested across five architectures.
- Honest benchmarking: multiple well-tuned Quicksort baselines, and the paper freely reports where Funnelsort loses (small  $N$ , Pentium 4, MIPS, and disk vs. TPIE).

- Error on page 6? — the buffer size is defined as  $\alpha \lceil d^{3/2} \rceil$ , where  $d$  is a fixed parameter. This doesn't make any sense - the whole point of the algorithm is that the buffers are not constant. It probably should be  $\alpha \lceil k^{d/2} \rceil$ .
- They observe that the asymptotic cache complexity is independent of layout and only depends on buffer sizes, but doesn't explain why.
- Only wall-clock time is measured; no hardware performance counters (cache-miss counts) to directly confirm the actual cache behavior

## Future directions

- Re-run the engineering study on modern hardware (multi-core, shared LLC, NVMe) and modern compilers.
- Extend to parallel / multi-core cache-oblivious sorting
- Use hardware performance counters to directly validate the cache-miss explanations, instead of inferring them from wall-clock trends.

Questions?

## Key references i

- Aggarwal, A. and Vitter, J. S. *The Input/Output Complexity of Sorting and Related Problems*. CACM, 1988.
- Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. *Cache-Oblivious Algorithms*. FOCS, 1999.
- Brodal, G. S. and Fagerberg, R. *Cache Oblivious Distribution Sweeping*. ICALP, 2002.
- Brodal, G. S. and Fagerberg, R. *On the Limits of Cache-Obliviousness*. STOC, 2003.
- Bentley, J. L. and McIlroy, M. D. *Engineering a Sort Function*. Software–Practice and Experience, 1993.
- Arge, L., Chase, J., Vitter, J. S., and Wickremesinghe, R. *Efficient Sorting Using Registers and Caches*. ACM JEA, 2002.

- LaMarca, A. and Ladner, R. E. *The Influence of Caches on the Performance of Sorting*. J. Algorithms, 1999.
- Department of Computer Science, Duke University. *TPIE: A Transparent Parallel I/O Environment*. 2002.

- I used Claude AI to help generate my slides.