Engineering a Cache-Oblivious Sorting Algorithm

Gerth Stølting Brodal, Rolf Fagerberg, Kristoffer Vinther

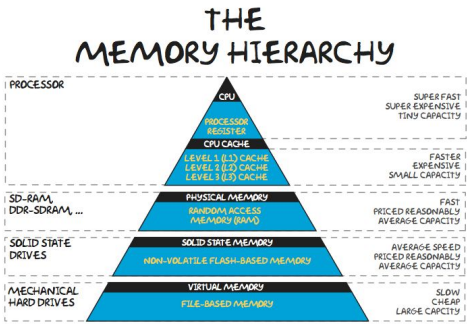
Agenda

- Cache-oblivious basics and the I/O model: why memory movement dominates runtime
- The Funnelsort idea and the Lazy Funnelsort variant: merger tree, buffers, and parameters (d, α)
- Engineering choices and experimental results (what actually made it fast)
- Pros, cons, and future directions

Why memory traffic dominates

Real machines are hierarchical: registers → L1/L2/L3 → DRAM → SSD/HDD; each level is slower and larger

- Algorithmic cost often hinges on access patterns, not just comparisons or arithmetic
- Sorting touches lots of memory with potentially poor locality; reducing transfers can dominate speed



External Memory Model

- Parameters: B = block size (items moved per transfer), M = fast memory capacity (items)
- Cost metric: # of block transfers between slow and fast memory; scanning N items costs $\Theta(N/B)$ I/Os
- Goal for sorting: minimize I/Os yet keep $O(N \log N)$ comparisons

Cache Oblivious Algorithms

- Write the algorithm without M or B in the code; analyze it for all M, B (optimal replacement assumed)
- Optimize for one unknown level → get multilevel optimality (L1, L2, L3, RAM, disk) “for free”
- Classic examples: cache-oblivious FFT, matrix transpose, and Funnelsort

Lower bound

• Any comparison sort in the I/O model needs Ω((N/B) · log_{M/B}(N/B)) block transfers

• Intuition: each “global pass” over the data costs Θ(N/B); with memory M you can multi-merge ≈ M/B runs per pass

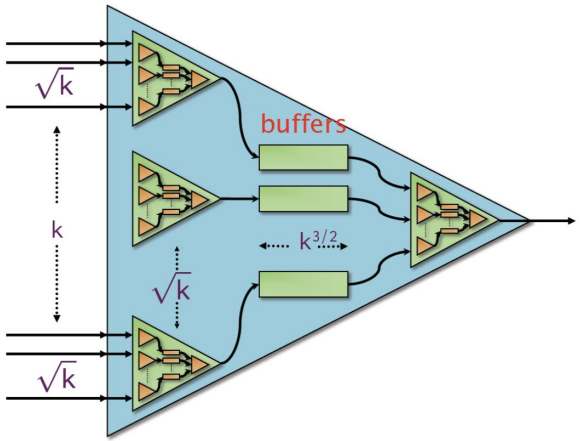
Tall Cache Assumption

- Classic Funnelsort analysis needs $M \geq B^2$ so buffers and working sets fit cleanly without thrashing
- Lazy Funnelsort relaxes to $M \geq B^{1+\varepsilon}$ for any fixed $\varepsilon > 0$, at the cost of only a constant $O(1/\varepsilon)$ factor
- Read it as: the cache can hold at least B^{ε} distinct blocks at once

Funnelsort → Lazy Funnelsort

• Funnelsort is a cache-oblivious multiway mergesort built from a merger tree with buffers on edges, k inputs split into sqrt(k) sets of sqrt(k) elements and producing sqrt(k) buffers sorted recursively

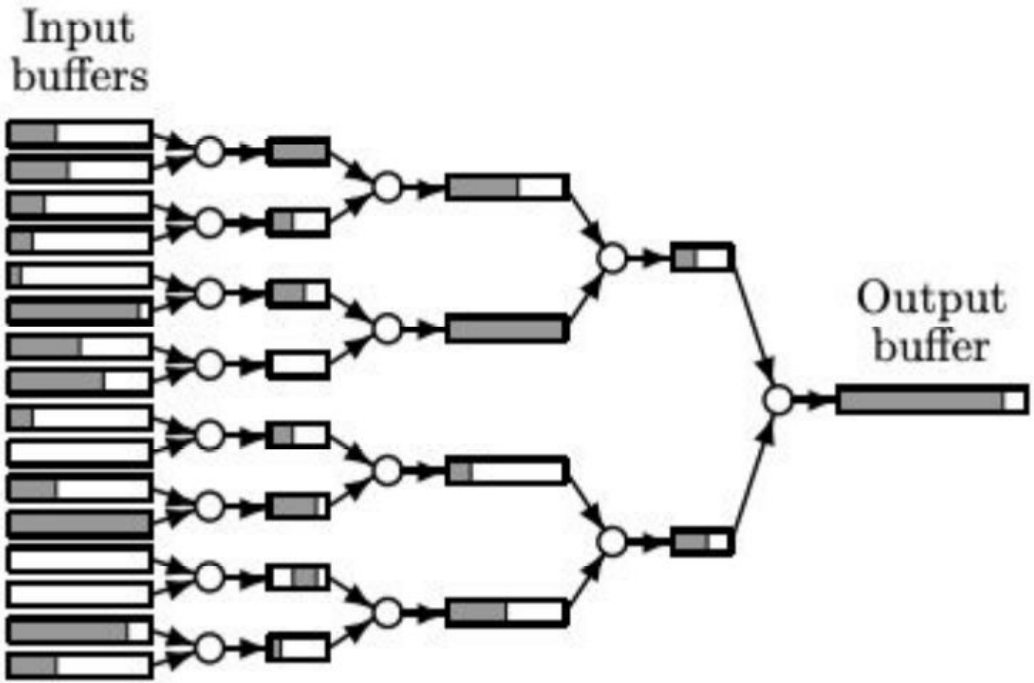
• Lazy Funnelsort keeps the asymptotics but uses on-demand (“fill-on-empty”) scheduling and generalizes ex



K-merger

- A k-merger is a perfect binary tree of k−1 binary mergers with input buffers at leaves and an output buffer at the root
- The root’s output buffer has size k^d (d > 1); one call emits Θ(k^d) items before needing another refill
- k is the number of input streams this structure merges

K-merger



Lazy scheduling (“fill-on-empty”)

- Instead of global scheduling, a node simply fills its output buffer when empty, pulling recursively from children
- This on-demand policy preserves streaming access and avoids machine-specific tuning
- It’s the main difference from the classic, more eager Funnelsort

```
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
```

Where the split sizes come from (N^{1/d}, N^{1−1/d})

- To output all N items in one top call, set k^d ≈ N ⇒ k = N^{1/d}
- Each of the k inputs must then contribute N/k = N^{1−1/d} items
- So the outer recursion is: sort N^{1/d} chunks of size N^{1−1/d}, then k-merge

Why multiway merging wants Θ(M/B) streams

- Smooth merging needs one input block per stream in cache, plus one output block
- If the working set fits in M, you can keep ≈ M/B streams “open” without evicting each other
- This lets a cache-oblivious sorter behave like an optimally tuned mergesort at the depth that fits in cache

How cache-obliviousness “just happens”

• The recursion decomposes the problem so that at some depth, a subproblem fits the current cache level

• At that depth you effectively get a Θ(M/B)-way merge without hard-coding M or B

• Repeat per level of the hierarchy → Θ(N/B) I/Os per level and log_{M/B}(N/B) effective levels overall

Quicksort takes Θ(N/Blog(N/M))

What the parameter d controls

- d is the coarseness exponent: larger d → bigger emission per root call (k^d) and larger internal buffers
- d also shapes the recursion (fewer but larger subproblems as d increases)
- In practice, d ≈ 2 works well; changing d shifts constants more than asymptotics

What the parameter α controls

- α is a constant multiplier in the buffer-size formulas; increasing α uniformly fattens buffers, decreasing α thins them
- α also sets the base-case cutoff (when to switch to a standard RAM sorter)
- Too small α shrinks buffers so much that refill overhead dominates; performance drops sharply

Why this yields Θ(M/B) effective fan-in

• At the depth where the active node's working set (one block per input stream + one output block) fits in M, it can keep ≈ M/B streams open

• That node therefore behaves like an (M/B)-way external mergesort for that cache level, even though the code never sets M or B

• Why "some depth" is enough: caches are inclusive and the recursion spans many granularities, so whichever depth matches the current cache level gives you a full Θ(N/B) sequential pass for that level; summing over levels yields the overall Θ((N/B) · log_{M/B}(N/B)) bound

I/O complexity

- Each effective pass reads and writes everything once → Θ(N/B) I/Os
- Each pass reduces runs by ≈ M/B → need log_{M/B}(N/B) passes total
- Multiply to get Θ((N/B) · log_{M/B}(N/B)); Lazy retains this with only a constant O(1/ε) factor under M ≥ B^{1+ε}

The “basic merger” degree z

- Base nodes can merge z inputs (not just 2); larger z reduces tree height (fewer element moves)
- But larger z increases per-step comparison/overhead; there’s a sweet spot
- Empirically, z ≈ 4 or 5 worked best

The inner merge loop

- Tried clever loops (e.g., “min-batching”, Hwang–Lin for skew) vs. a plain compare-and-move loop
- Fancy variants added overhead; the plain loop was consistently fastest
- Min-batching was typically 15–45% slower; non-hybrid Hwang–Lin could be ~3× slower

Merger layout and navigation

- Tested BFS, DFS, and van-Emde-Boas (vEB) layouts; pointer-based vs. implicit indexing; recursive vs. iterative calls
- Best combo: recursive + pointer-based + vEB layout + nodes/buffers stored separately
- Gains were measurable (~10% for vEB); worst combos could be ~65% slower

Merger caching

- Many mergers at a level share sizes; prebuild and reuse them, resetting between uses
- Yields a consistent 3–5% speedup with minimal code complexity

Base-case sorter

- Evaluated insertion/selection/heap/shell/quicksort for the base case
- Insertion sort often wins at very small sizes; practical builds switch to a tuned std::sort around a small threshold
- This matters because base cases are numerous in a deep recursion

Tuning the parameters α and d

- α fattens/thins buffers; d sets the burst size (k^d) and recursion grain
- Too small α or d ⇒ tiny buffers, frequent refills, poor amortization (big slowdowns)
- Sensible picks: α ≈ 16, d ≈ 2 on tested machines

The final engineered variants they benchmark

- Funnelsort2: $z = 2$, $(\alpha, d) = (16, 2)$, with a small-n base-case switch
- Funnelsort4: $z = 4$ with the same α, d; typically faster due to fewer levels and fewer total moves
- These were compared to tuned Quicksorts and cache-aware mergesorts

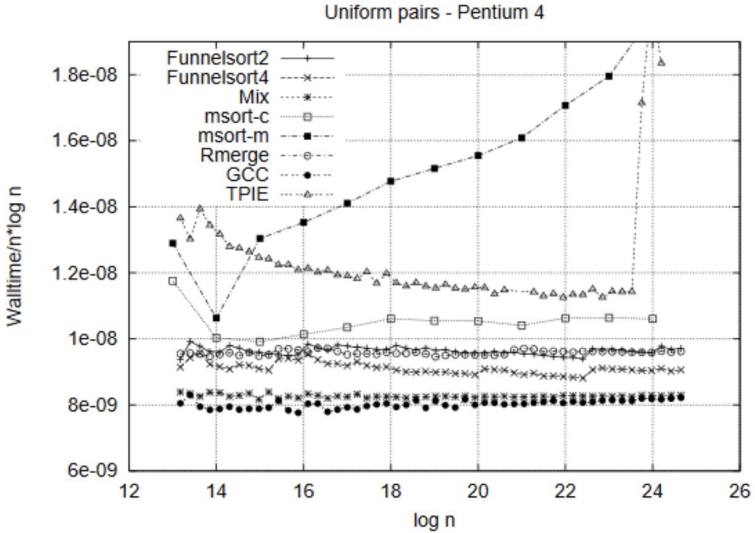
Experimental setup

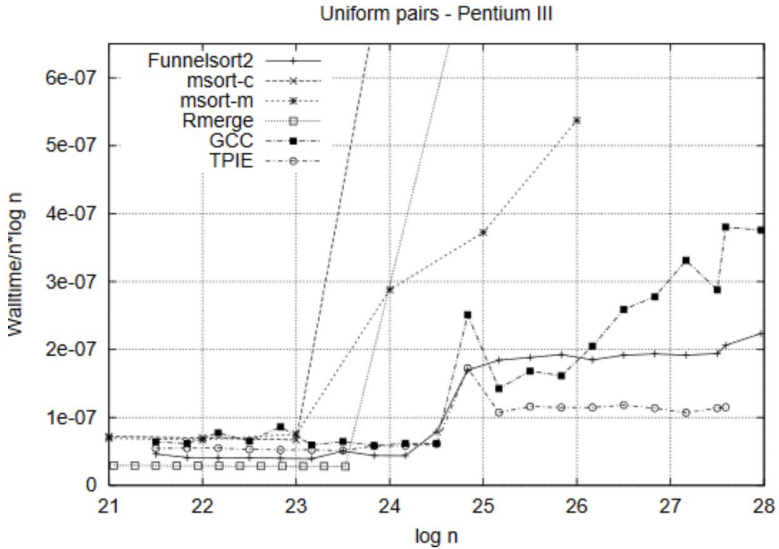
- Multiple architectures; measured wall-clock time
- Data types: integers; (int, pointer) pairs; 100-byte records
- Inputs: uniform, few-distinct-keys, and nearly-sorted datasets

Table I. 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	2400 MHz	800 MHz	175 MHz	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

Results


Uniform pairs - Pentium 4
Funnelsort2
Funnelsort4
Mix
msort-c
msort-m
Rmerge
GCC
TPIE
Walltime/n*log n
1.8e-08
1.6e-08
1.4e-08
1.2e-08
1e-08
8e-09
6e-09
12
14
16
18
20
22
24
26
log n


Uniform pairs - Pentium III
Funnelsort2
msort-c
msort-m
Rmerge
GCC
TPIE
Walltime/n*log n
6e-07
5e-07
4e-07
3e-07
2e-07
1e-07
0
21
22
23
24
25
26
27
28
log n

RAM Results

For small inputs, the best-engineered Funnelsort trails GCC std::sort by ~10–40%.

As n grows on three architectures, Funnelsort's cache behavior kicks in: it overtakes and ends up ~10–40% faster on the largest in-RAM instances.

TPIE (external memory designed) and other cache-aware algorithms performed worse

Exceptions:

MIPS R10000: slower CPU ⇒ CPU cycles dominate, masking cache benefits.

PC800/RDRAM system: large cache-line size reduces cache-latency impact during sequential scans, so cache advantages matter less.

Takeaway: On architectures where memory traffic—not CPU—dominates, the theoretically better cache performance of a well-tuned Funnelsort shows up in practice.

Disk Results

TPIE is the clear winner: purpose-built for external memory and engineered for disk.

Its double-buffering overlaps I/O and computation, an advantage that's hard to replicate in a cache-oblivious design, giving TPIE a decisive edge.

Lazy Funnelsort places second and clearly outperforms GCC std::sort in on-disk settings.

Funnelsort's gap over GCC grows with n, consistent with the log-base differences in I/O complexity between the approaches.

Algorithms tuned for CPU caches (RAM) perform notably poorly on disk, where sequential I/O and overlapping transfers matter most.

Conclusions

- Cache-oblivious sorting can be practically competitive with top Quicksorts in RAM and remains sound when spilling
- The merger tree + recursive buffer sizing (with d, α) yields long sequential runs and automatic Θ(M/B) fan-in at the right depth
- Engineering details (layout, inner loop, base case, degree z) matter a lot for constants

Pros, Cons, Future Directions

Pros:

- Extensive results with testing on multiple cases and architectures
- Good (atleast second-best) performance in both cases

Cons:

- Sequential search instead of grid search over parameters

Future Directions: Parallelism, dynamic parameters, tuning for specific data distributions