Motivation

- Memory hierarchy has become a fact of life
- Accessing non-local storage may take a very long time
- Good locality is important to achieving high performance

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Relative to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0.5 ns</td>
<td>1</td>
</tr>
<tr>
<td>L1 cache</td>
<td>0.5 ns</td>
<td>1-2</td>
</tr>
<tr>
<td>L2 cache</td>
<td>3 ns</td>
<td>2-7</td>
</tr>
<tr>
<td>DRAM</td>
<td>150 ns</td>
<td>80-200</td>
</tr>
<tr>
<td>TLB</td>
<td>500+ ns</td>
<td>200-2000</td>
</tr>
<tr>
<td>Disk</td>
<td>10 ms</td>
<td>$10^7$</td>
</tr>
</tbody>
</table>
Cache-Oblivious Model

Frigo, Leiserson, Prokop, Ramachandran, FOCS'99

- Program in the RAM model
- Analyze in the I/O model for arbitrary $B$ and $M$
- Optimal off-line cache replacement strategy
Cache-Oblivious Model

Frigo, Leiserson, Prokop, Ramachandran, FOCS'99

- Program in the RAM model
- Analyze in the I/O model for arbitrary $B$ and $M$
- Optimal off-line cache replacement strategy

- Optimal on arbitrary level $\implies$ optimal on all levels
- Portability
## Sorting — I/O Bounds

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickSort</td>
<td>$O \left( \frac{N}{B} \cdot \log_2 \frac{N}{M} \right)$</td>
<td></td>
</tr>
<tr>
<td>Binary MergeSort</td>
<td>$O \left( \frac{N}{B} \cdot \log_2 \frac{N}{M} \right)$</td>
<td></td>
</tr>
<tr>
<td>$\Theta \left( \frac{M}{B} \right)$-way MergeSort</td>
<td>$O \left( \text{Sort}_{M,B}(N) \right)$</td>
<td>Aggarwal and Vitter 1988</td>
</tr>
<tr>
<td>(Lazy) Funnelsort</td>
<td>$O \left( \text{Sort}_{M,B}(N) \right)$</td>
<td>Frigo, Leiserson, Prokop and Ramachandran 1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Brodal and Fagerberg 2002</td>
</tr>
</tbody>
</table>

- ★ cache-aware
- ★ cache-oblivious
- ★ requires $M \geq B^{1+\varepsilon}$

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

Engineering a Cache-Oblivious Sorting Algorithm
Lazy Funnelsort

Divide input in $N^{1/3}$ segments of size $N^{2/3}$
Recursively Funnelsort each segment
Merge sorted segments by an lazy $N^{1/3}$-merger

Brodal and Fagerberg 2002
Lazy $k$-merger

Buffer size $\alpha \cdot \sqrt{k}^d$

Brodal and Fagerberg 2002
Engineering Lazy Funnelsort

- Recursive implementation beats iterative implementation
- 4- or 5-way merge beats 2-way merge
- Standard memory allocator beats hand-coded allocator
- Pointer based van Emde Boas layout beats implicit layouts
- Nodes and buffers stored separately beats one layout
- Straightforward beats hand-coded branch elimination in core loop
- $d = 2.5$ and $\alpha = 16$
- Reuse merger data structures
- For $k$-mergers of height $< 2$ switch to Quicksort
Evaluating Funnelsort

- 2-way and 4-way Funnelsort
- Quicksort
  - STL GCC
  - STL Intel C++
  - Sedgewick
  - Bentley & MacIlroy with pivot tuning
- TPIE - optimized for external memory
- Rmerge - optimized for registers
- Mergesort by LaMarca and Ladner
<table>
<thead>
<tr>
<th>Hardware Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture type</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Operation system</strong></td>
</tr>
<tr>
<td><strong>Clock rate</strong></td>
</tr>
<tr>
<td><strong>Address space</strong></td>
</tr>
<tr>
<td><strong>Pipeline stages</strong></td>
</tr>
<tr>
<td><strong>L1 data cache size</strong></td>
</tr>
<tr>
<td><strong>L1 line size</strong></td>
</tr>
<tr>
<td><strong>L1 associativity</strong></td>
</tr>
<tr>
<td><strong>L2 cache size</strong></td>
</tr>
<tr>
<td><strong>L2 line size</strong></td>
</tr>
<tr>
<td><strong>L2 associativity</strong></td>
</tr>
<tr>
<td><strong>TLB entries</strong></td>
</tr>
<tr>
<td><strong>TLB associativity</strong></td>
</tr>
<tr>
<td><strong>TLB miss handling</strong></td>
</tr>
<tr>
<td><strong>RAM size</strong></td>
</tr>
</tbody>
</table>

**Engineering a Cache-Oblivious Sorting Algorithm**
Comparison of Quicksort Implementations
Engineering a Cache-Oblivious Sorting Algorithm
Engineering a Cache-Oblivious Sorting Algorithm
Engineering a Cache-Oblivious Sorting Algorithm
Uniform pairs - Itanium 2

Walltime/n*\log n

GCC
Dink
Mix
Sedge

log n

12 14 16 18 20 22 24 26 28

5e-09 1e-08 1.5e-08 2e-08 2.5e-08 3e-08
Walltime/n*\log n vs \log n

- GCC
- Dink
- Mix
- Sedge

Uniform pairs - MIPS 10000

Engineering a Cache-Oblivious Sorting Algorithm
Results for Inputs in RAM
Engineering a Cache-Oblivious Sorting Algorithm
Engineering a Cache-Oblivious Sorting Algorithm
Engineering a Cache-Oblivious Sorting Algorithm
Engineering a Cache-Oblivious Sorting Algorithm

Walltime/n*\log n vs log n for Uniform pairs - MIPS 10000

- Funnelsort2
- Funnelsort4
- Mix
- msort-c
- msort-m
- Rmerge
- GCC

The graph compares the walltime per n*\log n for different sorting algorithms as the log of the input size n increases. The x-axis represents log n, while the y-axis shows walltime/n*\log n. The algorithms are plotted with distinct markers and line types to differentiate between them. The graph illustrates how each algorithm performs under varying input sizes, with GCC showing a distinct trend compared to the others.
Results for Inputs on Disk
Engineering a Cache-Oblivious Sorting Algorithm
Engineering a Cache-Oblivious Sorting Algorithm
Conclusion

- Very high performing generic sorting algorithm
- Performance remains robust across wide range of input sizes
- Across several different processor and operating system architectures
- On several different data types
- On several different input distributions
- *Overhead involved in being cache-oblivious can be small enough for the nice theoretical properties to actually transfer into practical advantages*