Hardware Caches and Optimization

Lukas Waymann

2017-06-27

Typical present-day CPUs have two or more levels of caches. This article provides basic insight into their operation and presents key architectural properties which suggest possible program optimizations.

The abstract external memory model (EMM) for memory hierarchies and the cache-oblivious model (COM) derived from it are presented briefly.

Contents

1 Introduction

A hardware cache is a comparatively fast and small physical memory. It stores a subset of the data present in slower, larger storage that is expected to be used again soon. The purpose of this additional memory is to reduce the number of accesses to the underlying slower storage.

There are fundamental reasons that having one single, uniform type of memory is not viable. No signal can propagate faster than the speed of light. Thus, every storage technology can only reach a finite amount of data within a desired access latency [12, p. 2].

The most ubiquitous example for hardware caches is the hierarchy of CPU caches that are found on almost all present-day CPUs. They are designated `L1` cache, `L2` cache, and so on, with `L1` being the fastest and smallest level. The underlying storage for CPU caches is the main memory.

There are more storage levels that comprise the *memory hierarchy* of a computer along with CPU caches and main memory. For example hard disk drives (HDDs) and solid-state drives (SSDs). However, swapping to HDDs and SSDs continues to become somewhat less common as main memory sizes increase. Even non-server systems can currently support 64 GiB of main memory, eliminating the need for swapping to disk under many workloads.

I will focus on how to use CPU caches effectively and the enabled performance gains in this article.

2 Motivation

Hardware caches are managed by hardware directly. They are generally opaque to the operating system and other programs. That is, software has no direct control over the contents of a hardware cache.

Despite this, two algorithms solving the same problem with the same asymptotic complexity (in the same $\Theta(g(n))$) may differ in performance by two orders of magnitude because of different *memory access patterns* [9]. We will see an example of this in section 6.

In a nutshell, hardware caches are ubiquitous but the performance gains they provide are conditional. Effective use of hardware caches requires knowledge about how they work. Algorithms must be designed and implemented observing this knowledge.

3 Cache Operation Overview

Whenever a program requests a memory address the CPU will search its caches. If the location is present, a *cache hit* occurs. Otherwise, the result is a *cache miss* and the next level of the memory hierarchy, which could be another CPU cache, is tried.

Unless explicitly prevented, the CPU brings all accessed data into cache (with some exceptions only relevant to OS programming) [3, p. 15]. This happens in response to cache misses and will, much more often than not, cause another cache entry to be *evicted* and replaced [3, p. 16].

4 Types of CPU Caches

Current x86 CPUs generally have three main types of caches: data caches, instruction caches, and translation lookaside buffers (TLBs) [13, 11:07]. Some caches are used for data as well as instructions and are called *unified*. [3, p. 20]. A processor may have multiple caches of each type, which are organised into numerical *levels* starting at 1, the smallest and fastest level, based on their size and speed.

In practice, a currently representative[1] x86 cache hierarchy consists of:

- Separate level 1 data and instruction caches of 32 to 64 KiB for each core (denoted `L1d` and `L1i` by Drepper [3, pp. 14–15]). Machine instructions in `L1i` are already decoded [3, pp. 31, 56].
- A unified `L2` cache of 256 to 512 KiB for each core.
- Often a unified `L3` cache of 2 to 16 MiB shared between all cores.
- One or more TLBs per core. They cache virtual-to-physical address associations of memory pages.

Estimates of typical access latencies are given by Ignatchenko [8].[2]

	`L1d`	`L2`	`L3`	Main Memory
Cycles	3–4	10–12	30–70	100–150

The biggest target for optimizations is the data cache. "[Instruction] cache is much less problematic" [3, p. 31] and optimizations for data and instruction cache tend to improve TLB usage as well [13, 11:53].

My laptop's AMD E-450 CPU has cores with an `L1d` cache of 32 KiB and a unified `L2` cache of 512 KiB each.[3] We can both verify these sizes and get a reasonably good measure of the access times by profiling listing 1 for different values of `SIZE`.[4] This program repeatedly reads elements from a thusly sized array in random order. To do this with minimal overhead, the array is first set up as a circular, singly linked list where every element except the last points to a random successor. When compiled with `-DBASELINE`, only this initialization is done.

We use random accesses because the CPU will detect and optimize sequential access by a technique called *prefetching* discussed in section 5.2, which would prevent us from determining access times.

[1]E.g. for AMD Family 14h processors [1, pp. 30–32], AMD Zen (17h) [17], and Intel Skylake desktop processors [10, figure 2-1, table 2-4]

[2]Intel [10, table 2-4], Meyers [13, 17:52, slide 18], Meyer, Sanders, and Sibeyn [12, pp. 2–3, 171], and Drepper [3, pp. 16, 20–21] all give comparable numbers for various architectures.

[3]Appendix A explains how to obtain this information.

[4]Appendix B.1 details how.

```c
#define N 100000000  // 100 million

struct elem {
   struct elem *next;
} array[SIZE];

int main() {
   for (size_t i = 0; i < SIZE - 1; ++i) array[i].next = &array[i + 1];
   array[SIZE - 1].next = array;
   // Fisher-Yates shuffle the array.
   for (size_t i = 0; i < SIZE - 1; ++i) {
      size_t j = i + rand() % (SIZE - i);  // j is in [i, SIZE).
      struct elem temp = array[i];  // Swap array[i] and array[j].
      array[i] = array[j];
      array[j] = temp;
   }
#ifndef BASELINE
   int64_t dummy = 0;
   struct elem *i = array;
   for (size_t n = 0; n < N; ++n) {
      dummy += (int64_t)i;
      i = i->next;
   }
   printf("%d\n", dummy);
#endif
}
```

Listing 1: Randomly Read Array Elements

Figure 1 shows the extra CPU cycles used by listing 1 in addition to the `BASELINE` version for different array sizes. That is, only the cycles used by the main loop are given, not those for initialization. I divided by `N` to get the cycles spent per loop iteration.

Up to 32 KiB, each access takes almost exactly 3 cycles.[5] This is the `L1d` access time. At 32 KiB (the size of the `L1d`) the time increases to about 3.4 cycles. This is not surprising since the cache is shared with other processes and the operating system, so some of our data gets *evicted*. The first dramatic increase happens at 64 KiB followed by smaller increases at 128 and 256 KiB. I suspect we are seeing a mixture of `L2` and `L1d` accesses, with less and less `L1d` *hits* and an `L2` access time around 25 cycles.

The values from 512 KiB (the size of the `L2`) to 128 MiB exhibit a similar pattern. The relative increase when the array size matches that of the `L2` is greater than for the `L1d` before; possibly because `L2` is a unified cache that also holds instructions. Eventually,

[5]The numerical results are shown in table 1 on page 17.

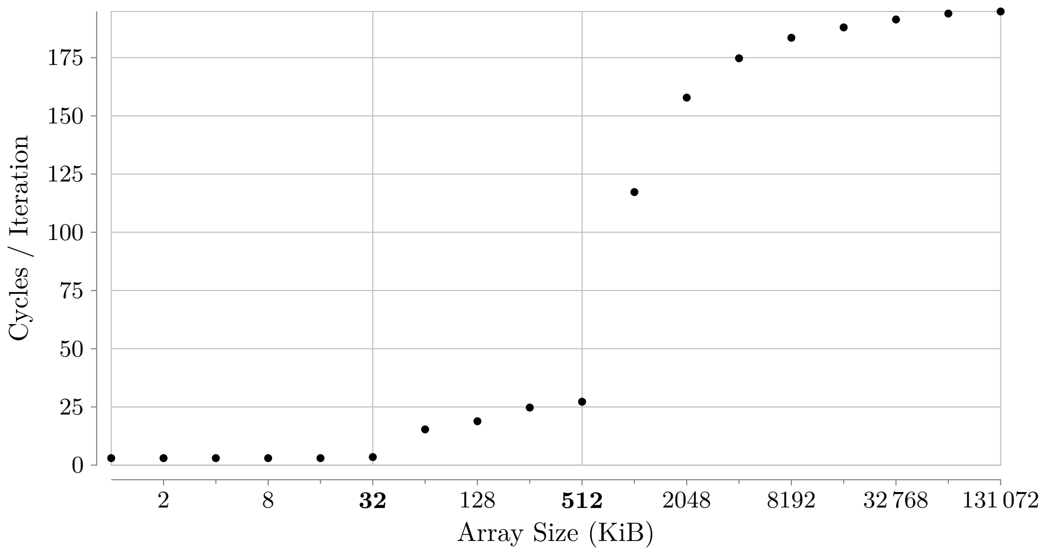

Figure 1: Access Times for Random Reads

more and more accesses go to main memory, causing delays of up to 200 cycles [cf. 3, p. 17, figure 3.4].

The data suggests that keeping the *working set* a process uses during a time interval small can yield dramatic performance improvements.

5 Key Concepts

Some architectural properties of hardware caches lead to important concepts for using them effectively.

5.1 Cache Line

Cache lines or *cache blocks* are the unit of data transfer between main memory and cache. They have a fixed size, which has been "64 bytes for many years" on x86/x64 CPUs [7, 13, 21:41].[6] This means accessing a single uncached 32-bit integer entails loading another 60 adjacent bytes.

My E-450 CPU is no exception and both of its data caches have 64-byte cache lines.[7] We can verify this quite easily. Consider listing 2. It loops over an array with an increment given at compile time as `STEP` and measures the processor time.

[6]Line sizes aren't *necessarily* identical among a CPU's caches. The Intel Pentium 4 processor had an `L1d` cache with "64 bytes per cache line" [6, p. 9] but an `L2` cache with "128 bytes per cache line" [6, p. 11].

[7]See Appendix A.

```c
#define SIZE 67108864  // 64 * 1024 * 1024.  The array will be 512 MiB.

int main() {
   int64_t* array = (int64_t*)calloc(SIZE, sizeof(int64_t));
   clock_t t0 = clock();
   for (size_t i = 0; i < SIZE; i += STEP) {
      array[i] &= 1;  // Do something.  Anything.
   }
   clock_t t1 = clock();
   printf("%d %f\n", STEP, 1000. * (t1 - t0) / CLOCKS_PER_SEC);
}
```

Listing 2: Loop over `array` with Increment `STEP`

The results for different values of `STEP` are plotted in fig. 2. As expected, the time roughly halves whenever the step size is doubled — but only from a step size of 16. For the first 4 step sizes, it is almost constant.

This is because the run times are primarily due to memory accesses. Up to a step size of 8, every 64-byte line has to be loaded. At 16, the values we modify are 128 bytes apart,[8] so every other cache line is skipped. At 32, three out of four cache lines are skipped, and so on [cf. 14, example 2].

Both cache and main memory can be thought of as being partitioned (in the set-theoretic sense) into cache lines. Data is not read or written starting from arbitrary main memory addresses, but only from addresses that are multiples of the cache line size.

5.2 Prefetching

Consider a simplified version of listing 1 that, instead of using random accesses, simply walks over the array sequentially. It still follows the pointers to do this, but the array is no longer shuffled. The results of profiling this new program as listing 1 before are plotted in fig. 3.[9]

Until the working set size matches that of the `L1d`, the access times are virtually unchanged at 3 cycles, but exceeding the `L1d` and hitting the `L2` increases this by no more than a single cycle. More strikingly, exceeding the `L2` has similarly limited effect. The access time plateaus not much above 6 cycles — about 3 % of the maximum we saw for random reads. Much of this can be explained by the improved use of cache lines: the penalty of loading a cache line is distributed among 8 accesses now. This could at best get us down to 12.5 %. The missing improvements are due to *prefetching*.

Prefetching is a technique by which CPUs predict access patterns and preemptively push cache lines up the memory hierarchy before the program needs them. This can

[8] 16 `int64_t` values of 8 bytes each

[9] Table 2 on page 17 shows the numerical results.

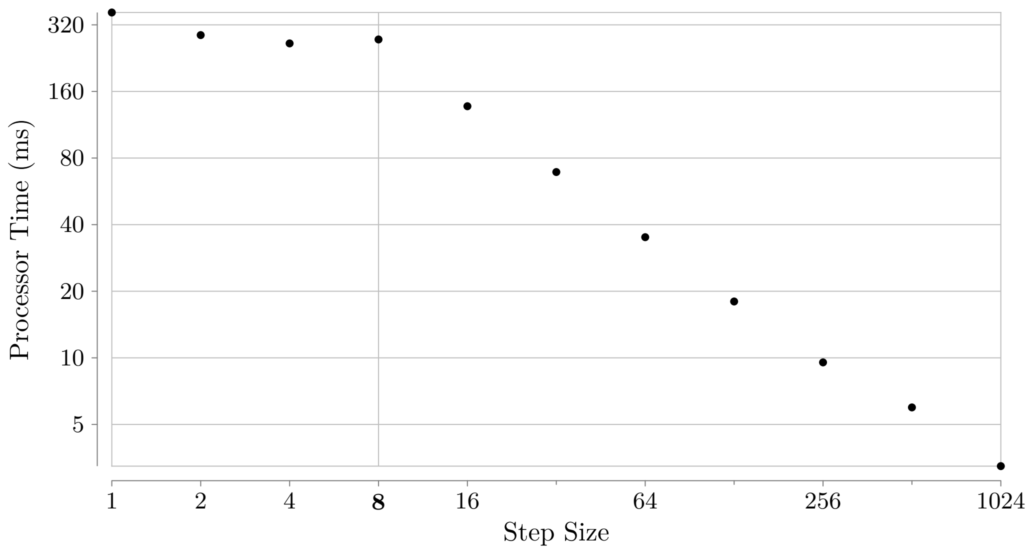

Figure 2: Processor Times for Running Listing 2

not work unless cache line access is predictable, though, which basically means linear [3, p. 60].[10]

Prefetching happens asynchronously to normal program execution [3, p. 14] and can therefore almost completely hide the main memory latency [3, p. 23]. This is not quite what we observe in fig. 3 because the CPU performs little enough work for memory bandwidth to become the bottleneck. Adding some expensive operations like integer divisions every loop iteration changes that and effectively levels the cycles spend per iteration across all working set sizes.[11]

What I described so far is *hardware* prefetching. It uses dedicated silicon to automatically detect access patterns. There is also *software* prefetching, which is triggered by special machine instructions that may be inserted by the compiler or manually by the programmer. Software prefetching is discussed in [3].

5.3 Locality of Reference

Two properties exhibited by computer code to varying degrees distinctly impact cache effectiveness: these are *spatial locality* and *temporal locality.* Both are measures of how well the code's memory access pattern matches certain principles.

[10]As an example, the most complicated *stride pattern* my laptop's CPU can detect is one that skips over at most 3 cache lines (for- or backwards) and may alternate strides (e.g. +1, +2, +1, +2, . . .) [1, p. 278].

[11]See fig. 6 on page 18.

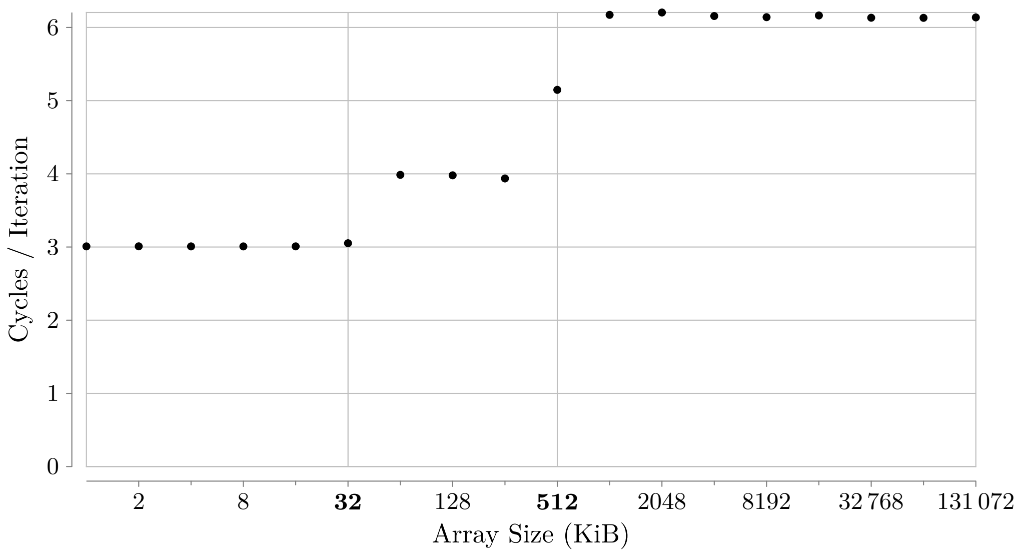

Figure 3: Access Times for Sequential Reads

5.3.1 Temporal Locality

One access suggests another. That is, once referenced memory locations tend to be used again within a short time frame. This is really the intrinsic motivation for having a memory hierarchy in the first place. When a cache line is loaded but not accessed again before being evicted, the cache provided no benefit.

5.3.2 Spatial Locality

1. For each accessed memory location, nearby locations are used as well within a short time frame. **2.** Memory is accessed sequentially. We have already seen in the last two sections that caches take advantage of both these principles by design:

1. Data is loaded in blocks; subsequent accesses to locations in an already loaded cache line are basically free.
2. Cache lines from sequential access patterns are prefetched ahead of time.

5.3.3 Notes

Access to instructions inherently has good spatial locality [3, p. 31] since they are executed sequentially outside of jumps, and good temporal locality because of loops and function calls [3, p. 14]. Programs with good locality are said to be *cache-friendly*.

6 Example: `std::vector` vs. `std::list`

The C++ program shown in listing 3, adapted from Ignatchenko [9], initializes a number of STL containers with random numbers and measures the processor time needed to sum all of them. I first ran it with `Container` being a type alias for `std::list`, then for `std::vector`.[12] Either way, the asymptotic complexity is $\Theta(N)$.

```cpp
constexpr int N = 5000;

int main() {
   Container containers[N];
   std::srand(std::time(nullptr));
   // Append an average of 5000 random values to each container.
   for (int i = 0; i < N * 5000; ++i) {
      containers[std::rand() % N].push_back(std::rand());
   }

   int sum = 0;
   std::clock_t t0 = std::clock();
   for (int m = 0; m < N; ++m) {
      for (int num : containers[m]) {
         sum += num;
      }
   }
   std::clock_t t1 = std::clock();

   // Also print the sum so the loop doesn't get optimized out.
   std::cout << sum << '\n' << (t1 - t0) << '\n';
}
```

Listing 3: Compute the Sum of a Container; adapted from Ignatchenko [9]

My result is that computing the sum runs 158 times faster when using `std::vector`. Some of this difference can be attributed to space overhead of the linked list and the added indirection, but the more cache-friendly memory access pattern of `std::vector` is decisive [9, pp. 5 sq.]. Using `std::list` incurs “random access to memory” in this example [9, p. 6].

6.1 Notes

6.1.1 “True” OO Style

In object-oriented (OO) systems, variables are typically referred to by pointers to a common base class. A polymorphic container of such pointers allows for dynamic dispatch of virtual functions. However, this carries the risk of degrading the performance of a sequential data structure to that of a list [16, 51:22]. See fig. 4.

[12]Each compiled with and `-O3` and `-march=native`

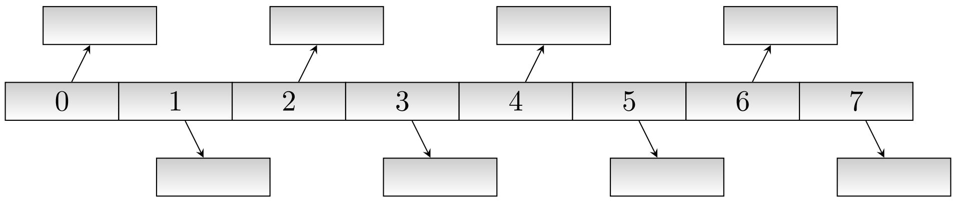

Figure 4: Array of Pointers; the array is compact but the actual objects may be scattered across memory pretty randomly.

7 Abstract?

We have seen that the hidden constant separating the time complexities of two reasonable algorithms under asymptotic analysis can get quite big in the presence of a memory hierarchy. To escape having to rely only on empirical results, abstract machine models taking the non-uniform memories of real-world computers into account can be used. One of these is the *external memory model (EMM)*.

7.1 External Memory Model

The EMM is an extension of the random access machine (RAM) model. While the latter assumes “a ‘sufficiently’ large uniform memory” [12, p. 5] with a constant access time, the EMM divides the memory into *internal* and *external*. The internal memory is accessed directly, but its size is limited to M items. The external memory is unbounded, but can only be accessed indirectly by loading data into internal memory “using I/Os that move B contiguous [items]” [12, p. 5].

The use of the term *I/O* here is somewhat non-standard. While it suggests external memory represents an HDD or SSD, it is not constrained which physical storages are associated with internal and external memory. If we choose the set of all CPU caches and the main memory, B becomes the cache line size and M may be in the order of a few MiB.

The number of I/Os an algorithm requires in the EMM can augment the information provided by standard asymptotic complexity analysis, but is no substitute for measurements. For example, the lower bound of I/Os needed for computing the sum of some input of size N, like in listing 3, is $(\lceil N/B \rceil + 1)$ in the EMM.[13] The linked list in that example probably takes almost N I/Os, though, since consecutive nodes are unlikely to fall into the same cache line. Thus, the predicted performance difference between `std::vector` and `std::list` is at most B, the number of items a cache line can hold, which is 16 in this case.[14] Recalling that the measured performance difference was 158, this is pretty inaccurate, but more informative than saying both data structure’s time complexities for traversal are $\Theta(N)$.

Even with the simplifications made by the EMM, algorithmic analysis is usually only

[13]The data could occupy a small amount of a cache line at its start and end.

[14]A cache line is 64 bytes on my laptop’s CPU and an `int` 4 bytes with my compiler.

done asymptotically: the number of I/Os is expressed in terms of $\mathcal{O}(f(N, M, B))$ or one of the related symbols. Theoretical lower bounds of the I/Os needed in the EMM are available in the literature (for example [12]) for many problems.

7.1.1 Limitations

While the concept of I/Os directly models cache lines, most other characteristics of memory hierarchies are ignored by the EMM. For example:

- prefetching, or more generally the advantages of sequential access patterns,
- multi-level caches,
- the lack of direct control over the contents of caches,
- associativity,[15]
- TLB.

More fundamentally, the model's premise is that I/Os are much more expensive than computation [12, p. 188]. While this is plausible when accessing HDDs, it doesn't apply to data transfer between main memory and caches to nearly the same extent. Some of these shortcomings are addressed by refined machine models, which include more details of real caches [12, p. 178]. This complicates mathematical analysis [12, p. 181] and heuristics may be used [12, p. 191], which in turn exacerbates the need for accompanying measurements [12, p. 181].

7.2 Cache-Oblivious Model

The *cache-oblivious model (COM)* or *ideal-cache model*, introduced by Frigo et al. [4], concedes most of the aforementioned problems to empirical evaluation and further increases the level of abstraction. Algorithms for the COM, called *cache-oblivious algorithms*, are designed without the cache size M or block size B as parameters. This seems silly since these values typically can be queried easily at both run and compile time [3, p. 50] but, perhaps counterintuitively, models one aspect of memory hierarchies better than the EMM. An algorithm that performs well in the COM performs well across the entire memory hierarchy [12, pp. 194 sq., 2, p. 4]; the same argument for data movement being *optimal* applies between any two levels of memory [5, lemma 15, p. 10].

Optimal means that the asymptotic [5, p. 2] number of cache misses incurred matches the problem's lower bound in the COM.

Cache misses take the place of I/Os because cache-oblivious algorithms don't manage the cache explicitly. This wouldn't be possible since the algorithms know neither the cache nor the cache line size [2, p. 5]. Instead, the COM uses the optimal replacement strategy of evicting the cache line that won't be accessed for the longest time in the future (Bélády's Algorithm). This is strangely at odds with real-world caches that don't know the future. However, Prokop proves that for many algorithms[16] cache misses only

[15]Associativity is not discussed in this paper; see Drepper [3] instead.

[16]Those satisfying equation (7.1) in [15, p. 46]. If the number of cache misses incurred by the algorithm only "depends polynomially on the cache size M", the equation is satisfied [2, p. 6].

increase by a constant factor when switching to a feasible replacement strategy [15, corollary 19, p. 46].[17]

Prokop further justifies the model by proving, "with only minor assumptions" [15, p. 12], that cache-oblivious algorithms that are optimal in the COM can by executed with an optimal amount of I/Os in the EMM [15, theorem 32, p. 56]. In other words, most cache-oblivious algorithms can be systematically transformed into cache-aware algorithms that asymptotically require the same amount of memory transfers in the EMM as the cache-oblivious variant in the COM.

7.2.1 Cache-Oblivious Matrix Transposition

The straightforward way to transpose an $m \times n$ matrix D out-of-place is to use two loops like so:

```c
for (int i = 0; i < m; ++i)
   for (int j = 0; j < n; ++j)
      E[j][i] = D[i][j];
```

Assuming D and E are stored in row-major layout (as they would be in the C and C++ languages), the reads from D are sequential memory accesses but the writes to E are not.

When D^T has sufficiently long rows $(m > B)$[18], every consecutive access will be to a different cache line. If it has sufficiently many rows, the constant amount of lines that are still kept in cache by the optimal replacement strategy once the inner loop completes and we need them again becomes negligible. Therefore, this algorithm incurs $\Theta(mn)$ cache misses.

The algorithm is by definition already cache-oblivious since it doesn't use M or B, but it is not optimal. We can do better with a divide-and-conquer approach. The idea is to recursively divide the input matrix D into two equal-sized submatrices along the greater dimension. If $m \geq n$ (more rows than columns), let

$$D = \begin{bmatrix} D_1 \\ D_2 \end{bmatrix}$$

and use $D^\mathsf{T} = \begin{bmatrix} D_1^\mathsf{T} & D_2^\mathsf{T} \end{bmatrix}$ to compute the transpose. If $m < n$, analogously slice D vertically. Eventually, the submatrices will become small enough so that pairs of input and output submatrices fit into cache at the same time, at which point it doesn't matter in what order we access the elements.[19] The recursion continues all the way down to 1×1 submatrices, but this doesn't change the theoretical analysis. According to Prokop [15, theorem 2 and 3, pp. 19–21] this algorithm incurs an optimal amount of $\Theta(1 + mn/B)$ cache misses. Listing 4 shows my implementation in C.

[17]e.g. LRU, FIFO, and random replacement

[18]D^T will be an $n \times m$ matrix, so m is its number of columns.

[19]No cache line will have to be evicted once loaded.

```
// Transpose the submatrix $(d_{ij})_{i \in I,\, j \in J}$.
void transpose(int I[2], int J[2], int D[m][n], int E[n][m]) {
   int num_rows = 1 + I[1] - I[0];
   int num_cols = 1 + J[1] - J[0];
   if (num_cols == 1 && num_rows == 1) {
      E[J[0]][I[0]] = D[I[0]][J[0]];
   } else if (num_cols <= num_rows) {
      // Horizontally slice D into two submatrices and recurse.
      transpose((int[2]){I[0], I[0] + num_rows / 2 - 1}, J, D, E);
      transpose((int[2]){I[0] + num_rows / 2, I[1]}, J, D, E);
   } else { /* Vertically slice D analogously... */ }
}
```

Listing 4: COM-Optimal Matrix Transposition

In practice, the new algorithm performs worse than the straightforward one for a lot of matrix sizes before it eventually pulls ahead. Figure 5 gives the speedups for square matrices of various sizes. The problem may be the excessive use of recursion.

Kumar works around this by “[stopping] the recursion when the problem size becomes less than a certain block size and then [using] the simple for loop implementation inside the block” [12, pp. 199–201]. This seems to compromise the cache-obliviousness of the algorithm. Perhaps cache-oblivious algorithms should be seen as a starting point for further optimizations that explicitly use the available information about the memory hierarchy of the machine a program runs on.

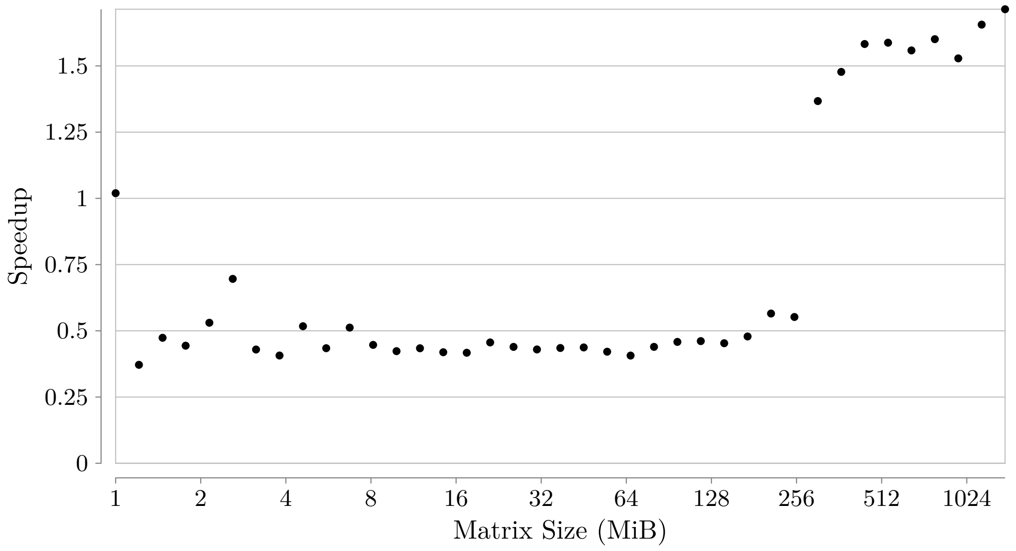

Figure 5: Speedup Achieved by COM-Optimal Matrix Transposition; the CPU time used by the straightforward matrix transposition algorithm divided by that used by the optimal one

A Reading Information About the CPU

There are many ways to display information about the processor(s) the operating system is running on. Among others, the `lscpu(1)` and `getconf(1)` programs and the `/proc/cpuinfo` pseudo-file on Linux. This is how I checked my CPU's cache sizes, for example:

```
$ lscpu | grep 'L1d\|L2'
L1d cache:              32K
L2 cache:               512K
```

This is what I used to get the cache line sizes:

```
$ getconf LEVEL1_DCACHE_LINESIZE; getconf LEVEL2_CACHE_LINESIZE
64
64
```

B Profiling Methodology

All programs were compiled and run on Linux using the GNU Compiler Collection (GCC)[20] with `-O2` or `-O3` and `-march=native`. I disabled CPU frequency scaling with `cpupower(1)`, reduced the number of running processes,[21] and assigned a high priority to the benchmark process with the `chrt(1)` command.

The *makefile* used to compile and run the measurements as well as the full source code of the programs is available at `https://github.com/meribold/cache-seminar-paper`.

B.1 Measuring CPU cycles

I used the `ocount(1)` event counting tool added to the *OProfile* [11] project in version 0.9.9. It uses *hardware performance counters*[22] to provide low-overhead performance information without source code modifications. Essentially, this is the procedure:

```
$ gcc -march=native -O2 program.c
# cpupower frequency-set -g performance
# chrt -f 99 ocount -e CPU_CLK_UNHALTED ./a.out
```

[20]in version 6.3.1

[21]I stopped Firefox (which added visible noise) and the X server, for example.

[22]`https://en.wikipedia.org/wiki/Hardware_performance_counter`

C Data

Array Size (KiB)	Cycles / Iteration	Array Size (KiB)	Cycles / Iteration
1	3.01	512	27.23
2	3.01	1024	117.28
4	3.01	2048	157.85
8	3.01	4096	174.74
16	3.01	8192	183.54
32	3.46	16 384	188.00
64	15.34	32 768	191.39
128	18.85	65 536	193.95
256	24.73	131 072	194.83

Table 1: Access Times for Random Reads

Array Size (KiB)	Cycles / Iteration	Array Size (KiB)	Cycles / Iteration
1	3.01	512	5.15
2	3.01	1024	6.17
4	3.01	2048	6.20
8	3.01	4096	6.16
16	3.01	8192	6.14
32	3.05	16 384	6.16
64	3.99	32 768	6.13
128	3.98	65 536	6.13
256	3.94	131 072	6.14

Table 2: Access Times for Sequential Reads

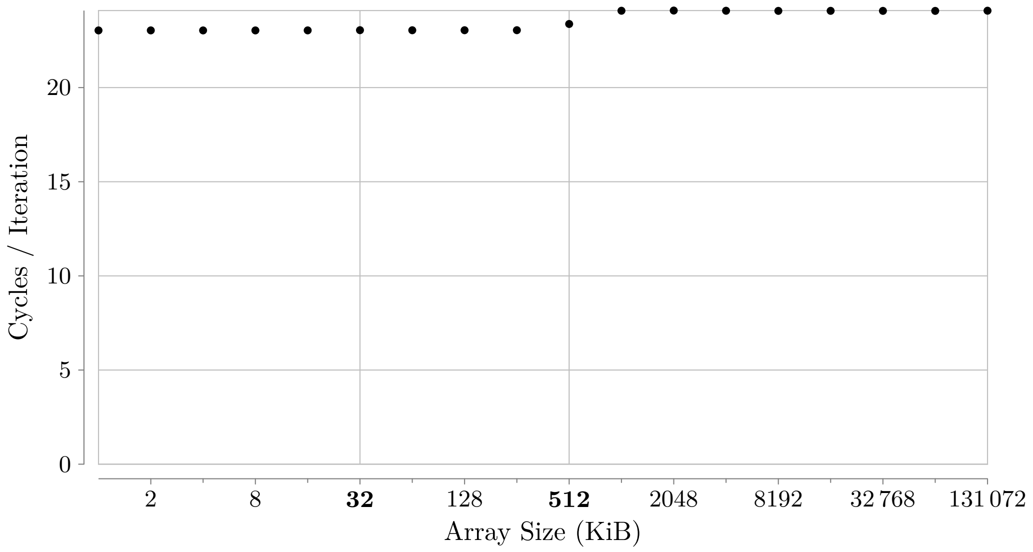

Figure 6: CPU-Bound Sequential Read Access

Acronyms

COM cache-oblivious model. 1, 12–15

EMM external memory model. 1, 11–13

GCC GNU Compiler Collection. 16

HDD hard disk drive. 3, 11, 12

I/O input/output operation. 11–13

OO object-oriented. 10

RAM random access machine. 11

SSD solid-state drive. 3, 11
STL Standard Template Library. 9

TLB translation lookaside buffer. 4, 12

References

[1] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 14h Models 00h-0Fh Processors*. Version 3.13. AMD. 2012-02-17. 470 pp. URL: `https://support.amd.com/TechDocs/43170_14h_Mod_00h-0Fh_BKDG.pdf`.

[2] Erik D. Demaine. "Cache-Oblivious Algorithms and Data Structures". In: *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, 2002-06-27/2002-07-01. URL: `http://erikdemaine.org/papers/BRICS2002/` (visited on 2017-06-16).

[3] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Ed. by Jonathan Corbet. 2007-11-21. 114 pp. URL: `https://people.redhat.com/drepper/cpumemory.pdf` (visited on 2017-05-15).

[4] Matteo Frigo et al. "Cache-Oblivious Algorithms. Extended abstract submitted for publication". 1999-05.

[5] Matteo Frigo et al. "Cache-Oblivious Algorithms". In: *40th Annual Symposium on Foundations of Computer Science*. IEEE. 1999-10-17/1999-10-19, pp. 285–297. DOI: `10.1109/SFFCS.1999.814600`.

[6] Glenn Hinton et al. "The Microarchitecture of the Pentium® 4 Processor". In: *Intel Technology Journal* 5.1 (Q1 2001). URL: `http://www.ecs.umass.edu/ece/koren/ece568/papers/Pentium4.pdf` (visited on 2017-05-13).

[7] Sergey Ignatchenko. *C++ for Games: Performance, Allocations and Data Locality*. 2016-05-23. URL: `http://ithare.com/c-for-games-performance-allocations-and-data-locality/` (visited on 2017-05-11).

[8] Sergey Ignatchenko. *Infographics: Operation Costs in CPU Clock Cycles*. 2016-09-12. URL: `http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/` (visited on 2017-05-19).

[9] Sergey Ignatchenko. "Some Big-Os are Bigger Than Others". In: *Overload* 134 (2016-08). Big-O notation is often used to compare algorithms. Sergey Ignatchenko reminds us that asymptotic limits might not be generally applicable., pp. 4–7. ISSN: 1354–3172. URL: `https://accu.org/var/uploads/journals/Overload134.pdf#page=6` (visited on 2017-05-11).

[10] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel. 2016-06. 672 pp. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[11] John Levon et al. *OProfile*. Comp. software. Version 1.1.0. 2015-08-03. URL: `http://oprofile.sourceforge.net`.

[12] Ulrich Meyer, Peter Sanders, and Jop Sibeyn, eds. *Algorithms for Memory Hierarchies. Advanced Lectures*. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer-Verlag, 2003. ISBN: 3-540-00883-7.

[13] Scott Meyers. "CPU Caches and Why You Care". Talk given at the code::dive conference. `http://codedive.pl/index/year2014`. Hala Stulecia, Wrocław, Poland, 2014-11-05. URL: `https://youtu.be/WDIkqP4JbkE` (visited on 2017-05-13).

[14] Igor Ostrovsky. *Gallery of Processor Cache Effects*. 2010-01-19. URL: `https://igoro.com/archive/gallery-of-processor-cache-effects/` (visited on 2017-05-15).

[15] Harald Prokop. "Cache-Oblivious Algorithms". MA thesis. Massachusetts Institute of Technology, 1999-06. URL: `http://supertech.csail.mit.edu/papers/Prokop99.pdf` (visited on 2017-06-16).

[16] Bjarne Stroustrup. *C++11 Style*. Talk given at the GoingNative 2012 conference. Redmond, WA, USA: Microsoft, 2012-02-02. URL: `https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style` (visited on 2017-06-19).

[17] *The "Zen" Core Architecture*. AMD. URL: `https://www.amd.com/en-gb/innovations/software-technologies/zen-cpu#Microarchitecture` (visited on 2017-05-19).