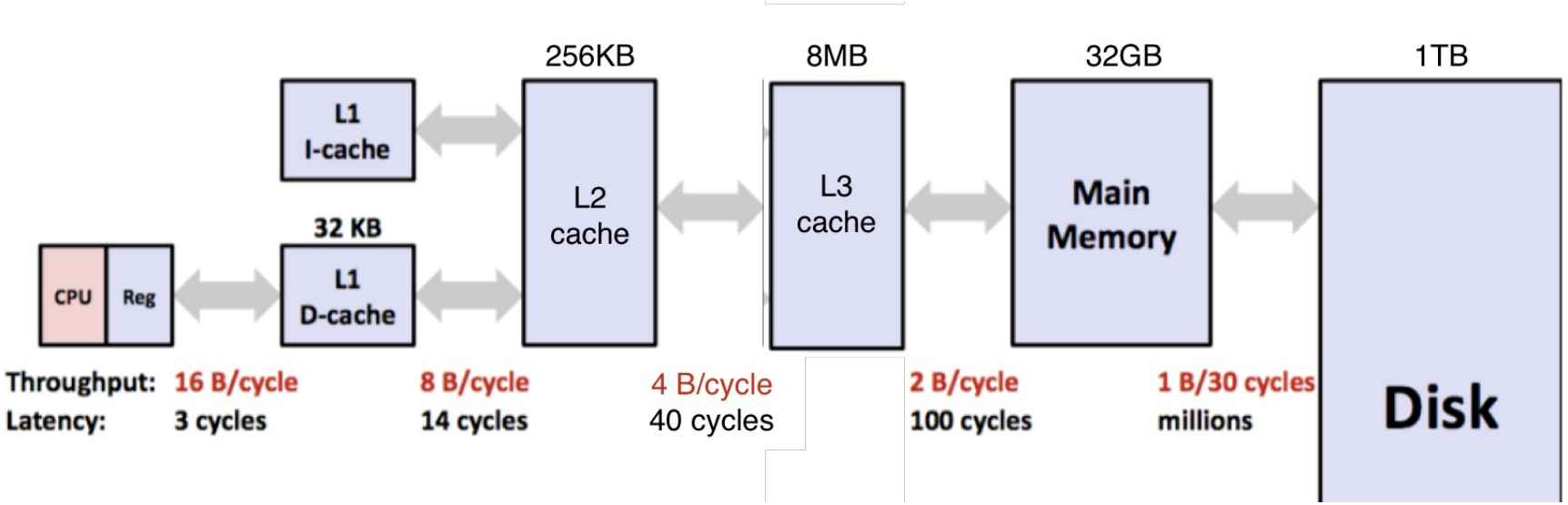
CS107, Lecture 25
Optimization, Caching, Writing Cache-Friendly Code

Reading: B&O 5

Ed Discussion: https://edstem.org/us/courses/28214/discussion/2250077

Caching

- Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
- Memory exists in levels and goes from **really fast** (registers) to **really slow** (disk).
- As data is more frequently used, it ends up in faster and faster memory.



Caching

All caching depends on locality.

Temporal locality

- Repeat access to the same data tends to be co-located in **time**
- Intuitively: things I have used recently, I am likely to use again soon

Spatial locality

- Related data tends to be co-located in **space**
- Intuitively: data that is near a used item is more likely to also be accessed

Caching

All caching depends on locality.

Realistic scenario:

- 97% cache hit rate
- Cache hit costs 1 cycle
- Cache miss costs 100 cycles
- How much of your memory access time is spent on 3% of accesses that are cache misses?

Demo: `cache.c`

Optimizing Your Code

- Explore various optimizations you can make to your code to reduce instruction count and runtime.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using callgrind
    - Optimize using –O2
    - And more…

Recap

- What is optimization?
- GCC Optimization
- Limitations of GCC Optimization
- Caching

Next time: wrap up

Lecture 24 and 25 takeaways: Compilers can apply various optimizations to make our code more efficient, without us having to rewrite code. However, gcc can only do so much! Sometimes we must optimize ourselves, using tools like `callgrind` and writing code to optimally leverage the cache hierarchy.