CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Memory usage
- Profiling Tools

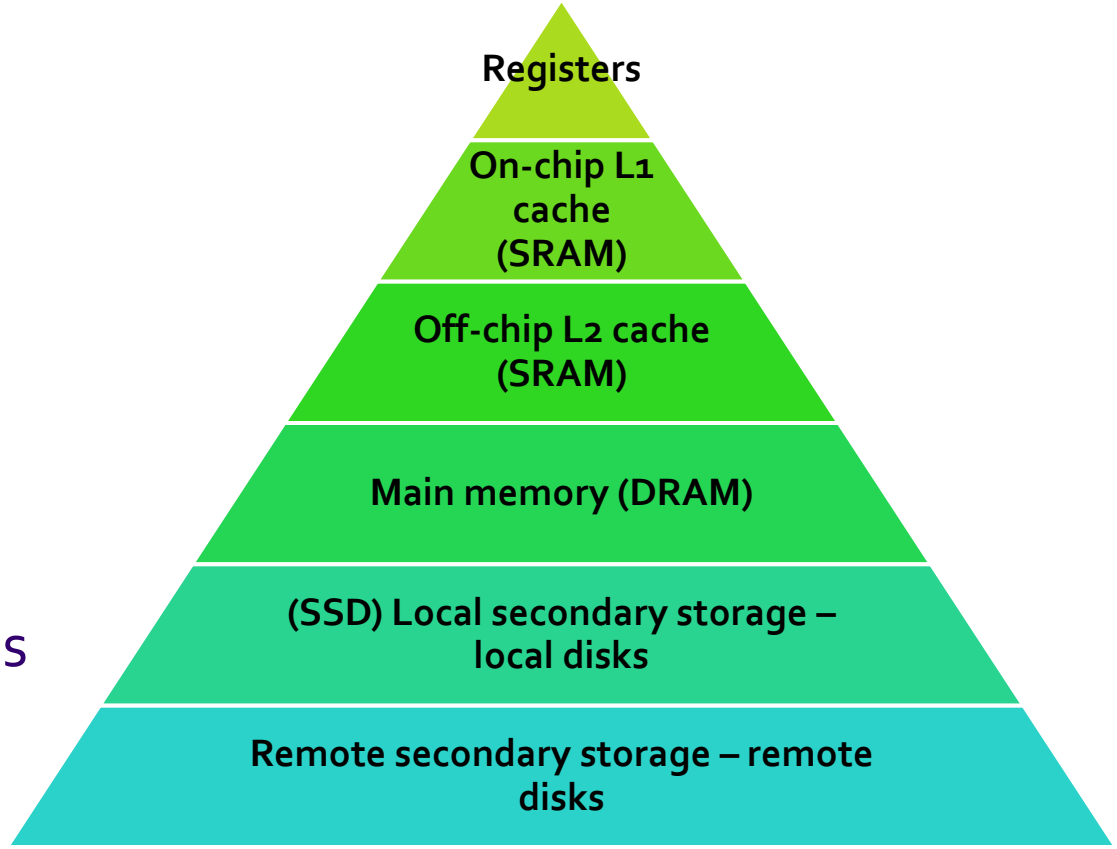
Your Goals

- Assessments
  - C Basics in-class Monday
  - Debugging demo
- Keep working on
  - HW5 (due Monday)
  - HW6 (due May 29)

Memory Hierarchy

- Smaller /Faster / Costlier
- Larger / Slower / Cheaper

- < 1 ns
- 1 ns
- 5-10 ns
- 100 ns
- 150,000 ns
- 1-150 ms

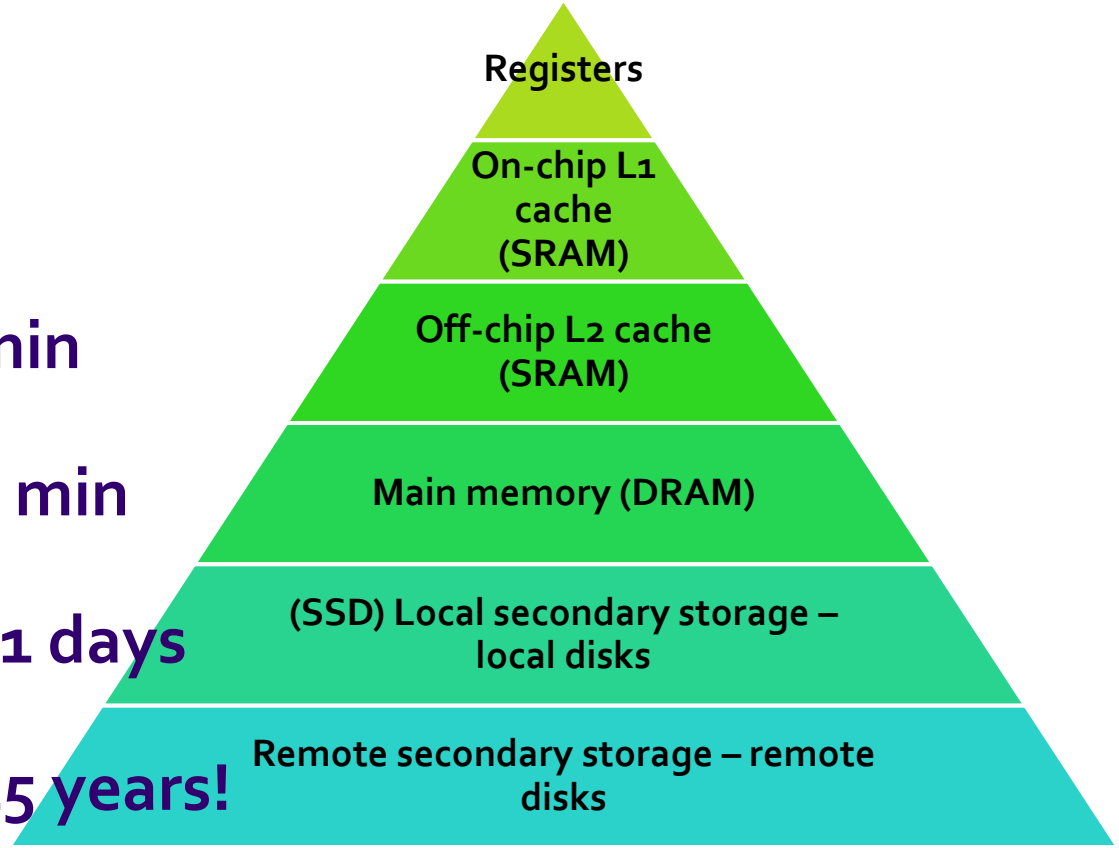


Memory Hierarchy - intuition

- Smaller /Faster / Costlier

- Larger / Slower / Cheaper

- < 1 ns – **5-10s**
- 1 ns – **20 s**
- 5-10 ns – **1-2 min**
- 100 ns – **15-30 min**
- 150,000 ns – **31 days**
- 1-150 ms – **1-15 years!**



Processor-Memory Bottleneck

- CPU speed has grown faster than bandwidth for data transfer
- Latency every time CPU has to go to the main memory
- 'cache' solution keeps recently accessed memory available
  - Loads a 'block' of memory at a time
    - Block size is architecture dependent. ~4 Kilobytes

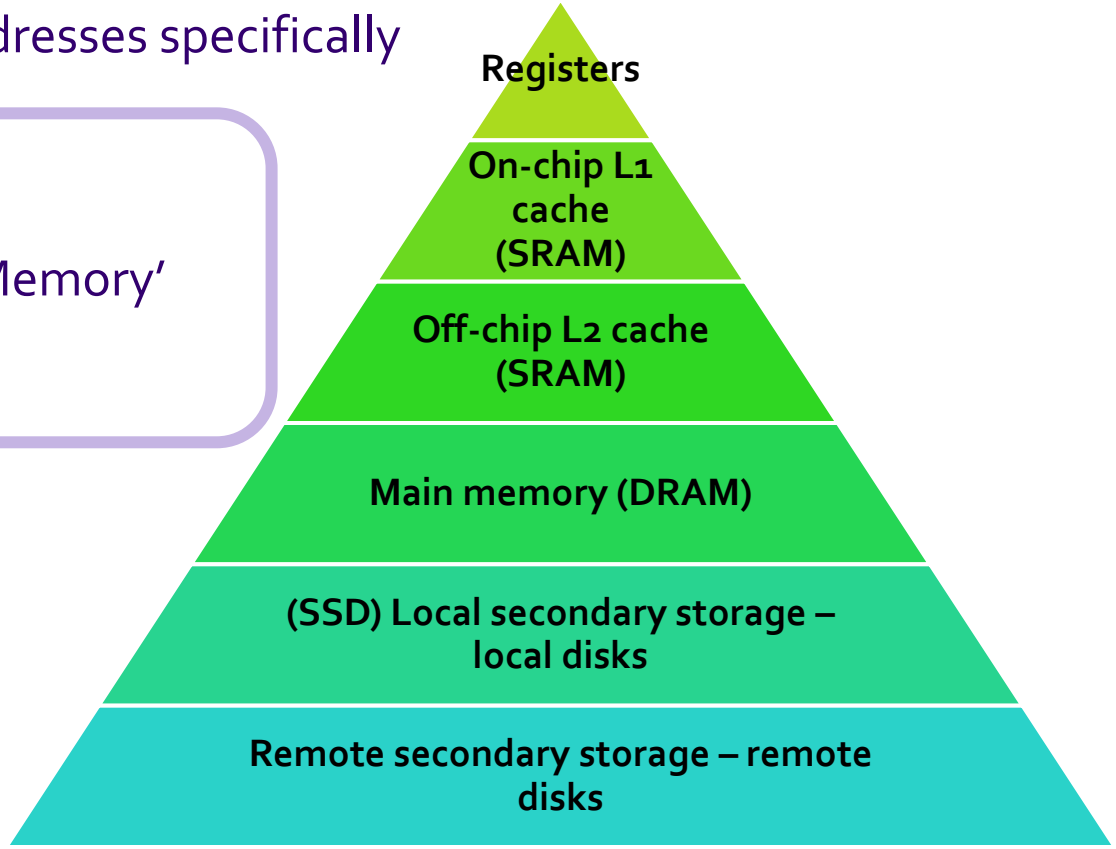
Memory Hierarchy - Caches

- Smaller /Faster / Costlier

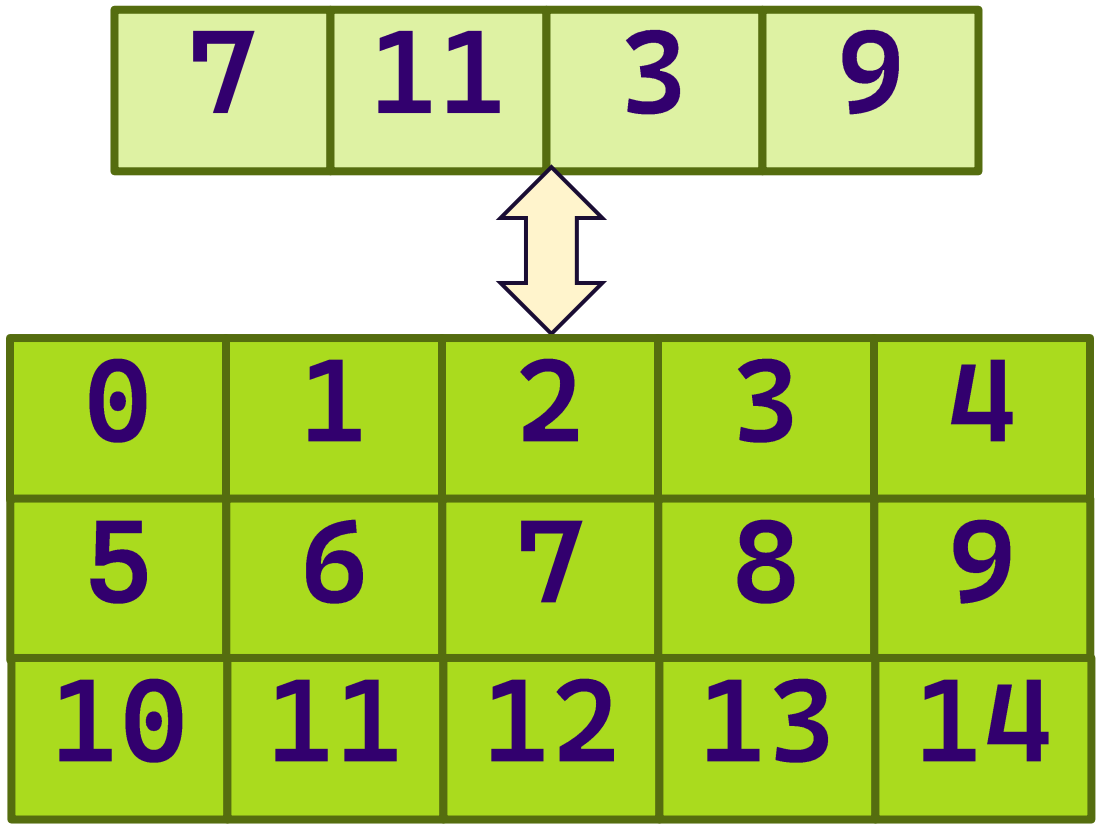
Program sees 'memory' but hardware manages transparently

- Larger / Slower / Cheaper

- Registers: program addresses specifically
- Cache L1
- Cache L2 'Program Memory'
- Main Memory
- 150,000 ns
- 10,000,000 ns
- 1-150 ms

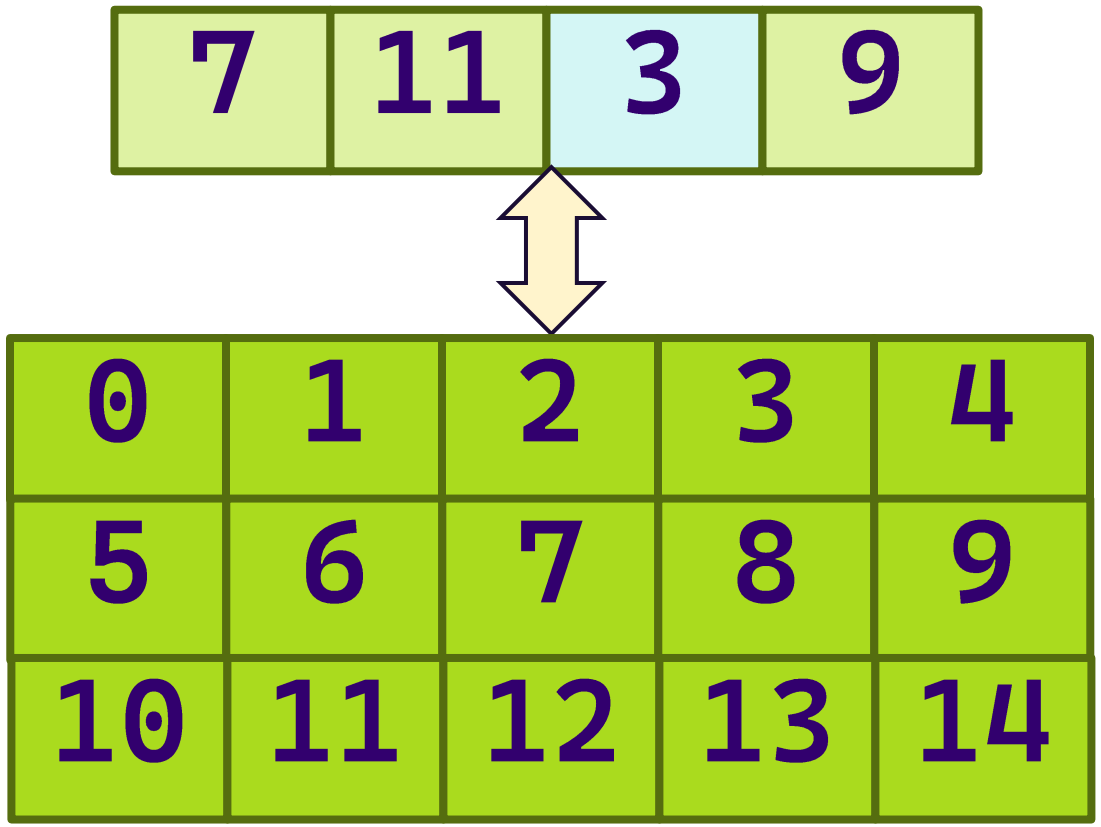


Cache Mechanics



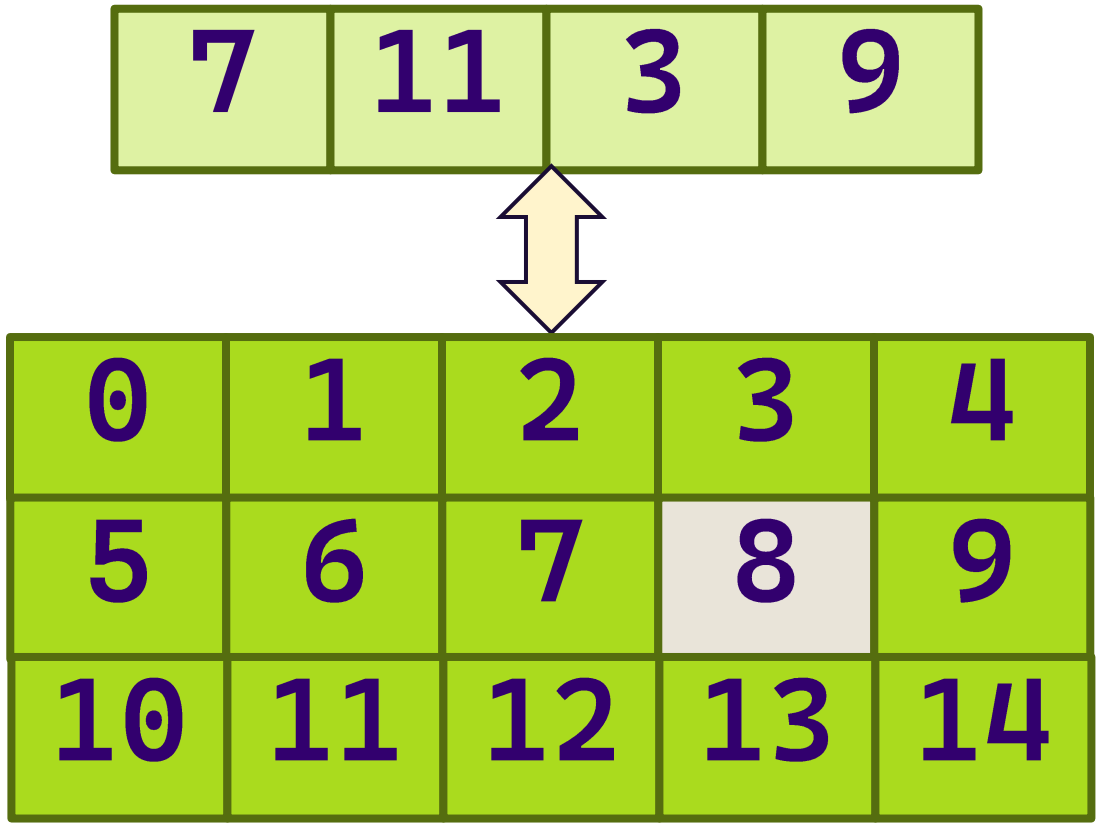
- Cache (smaller, faster, more expensive) holds subset of blocks
- Data is copied in block-sized units
- Memory (larger, slower, cheaper) can be considered divided into blocks

Cache Mechanics - retrieval



- If data from a block IN the cache is requested, it is easy to retrieve
- Data is sent to CPU

Cache Mechanics – retrieve from memory



- If data from a block **NOT IN** the cache is requested
- Data is **first** loaded from main memory to the cache
- **Second** it is passed to the CPU

Cache locality

Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

Temporal locality:

Recently referenced items are *likely* to be referenced again in the near future

Spatial locality:

Items with nearby addresses *tend* to be referenced close together in time

Analogy: took a bite of sandwich, probably going to take a bite out of other half of sandwich (as opposed to a new sandwich)

How do caches take advantage of this?

Locality

We consider locality in order to increase chances of a **hit** on the cache for memory retrieval

- *Temporal locality*: **`sum`** referenced every iteration
- *Spatial locality*: consecutive elements of **`array[]`** accessed

```
sum = 0;
for (i = 0; i < n; i++) {
            sum += a[i];
}
return sum;
```

Locality example (good)

Memory is accessed

`a[0][0], a[0][1], …`

…

`a[3][2], a[3][3]`

In other words, spatial locality.

```
int sum_array_rows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality example (bad)

Memory is accessed

`a[0][0], a[1][0], …`

…

`a[2][3], a[3][3]`

NO spatial locality.

```
int sum_array_rows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            sum += a[j][i];
    return sum;
}
```

Locality challenge

Can you improve temporal locality?

```
// Compute y = mx + b for each point
void compute_line(float m, float b, float*
  points, int n) {
   for(int i = 0; i < n; i++) {
     points[i] = m * points[i];
   }
   for(int i = 0; i < n; i++) {
     points[i] = b + points[i];
   }
}
```

Locality challenge solution

Can you improve temporal locality?

```
// Compute y = mx + b for each point
void compute_line(float m, float b,
 float* points, int n) {
  for(int i = 0; i < n; i++) {
    points[i] = m * points[i];
    points[i] = b + points[i];
  }
}
```

Writing Cache friendly code

Write code that has locality!

- Spatial: access data contiguously, use small strides
- Temporal: make sure access to the same data is not too far apart in time, keep working set reasonably small

How can you achieve locality?

- Adjust memory accesses in *code* (software) to improve miss rate (MR)
  - Requires knowledge of *both* how caches work as well as your system's parameters
- Proper choice of algorithm
- Loop transformations

Memory Summary

Memory Hierarchy

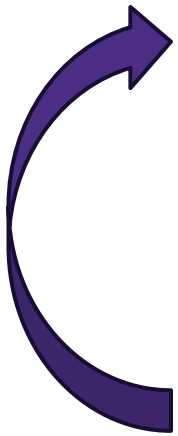
- Successively higher levels contain "most used" data from lower levels
- Accessing the disk is very slow
  - This is why we discourage excess I/O in homework assignments!
- Exploits *temporal and spatial locality*
- Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics

Cache Performance

- Ideal case: found in cache (hit)
- Bad case: not found in cache (miss), search in next level

Software specifications

- How do you know what software is supposed to do?
- Software engineering:
  - Requirements engineering
  - Specification writing & documentation
  - Architecture and design
  - Programming
  - Testing & Debugging
  - Deploying, operating, evolving



Effectiveness ?

- **Does what it is supposed to do**
- **Fails gracefully**
- **Uses memory safely and efficiently**
- **Computes in reasonable time**

Beyond Buglessness

Numerically reliable

Memory safe

Memory efficient

Operations efficient

Scales well

Benchmarks

- Compare performance against a standard measure
- Compare to industry standards
  - How are you doing compared to other solutions?
- Compare to previous performance
  - Is your implementation getting better?

- 3DMark (gamers hardware tests)
- MMLU (Language understanding)
- Frontier Math (AI math ability)
- Humanities Last Exam (multi-modal AI benchmark)

Basics

Write Code

Run Test Cases & Benchmarks

Profile code

Run linter for style (python clint.py)

Profiling Tools

- Investigate run-time behavior of code at different points
- Checks time taken by instructions from machine language to high-level functions
  - actual time
  - number of calls to the instruction
- Flat profiler - computes average call times, does not break down calls
- Call graph profiler - shows chains based on called functions

- Can use common benchmarks which exercise the code on tricky problems or corner cases
  - Goal of benchmark is to provide a standard test
  - Goal of profiler is to break down how the code is achieving its results

Insertion v Sampling profilers

Insertion

- Place specific profiling code in program
- Can be used on various platforms
- Accurate
- Requires recompilation and relinking
- Will affect performance

Sampling

- Monitoring or snap-shotting at specific intervals
- No modification of code
- Less accurate - limited by sampling rate
- Very small methods often missed
- Not great for memory

$gprof

Gnu profiling tool (insertion)

Compile with `$gcc -pg` flag

`$./mainopt`

Creates `gmon.out`

Run profiler with
`$gprof ./mainopt`

```
-----------------------------------------------

        0.00    0.06     102/102        main [1]
[2]   100.0    0.00    0.06     102        optimize [2]
        0.00    0.04 2040000/2040000    update_pos [4]
        0.02    0.00 2040000/2040000    update_vel [5]
        0.00    0.00     102/102        initialize_opt [6]
        0.00    0.00   51000/51000      update_gb [7]

-----------------------------------------------

        0.00    0.00     100/2008449    main [1]
        0.00    0.00    4000/2008449    initialize_opt [6]
        0.04    0.00 2004349/2008449    update_pos [4]
[3]    66.7    0.04    0.00 2008449        rastrigin [3]

-----------------------------------------------

        0.00    0.04 2040000/2040000    optimize [2]
[4]    66.5    0.00    0.04 2040000        update_pos [4]
        0.04    0.00 2004349/2008449    rastrigin [3]
        0.00    0.00   20085/20126      rosenbrock [8]
        0.00    0.00   20085/20126      spherefunc [9]
```

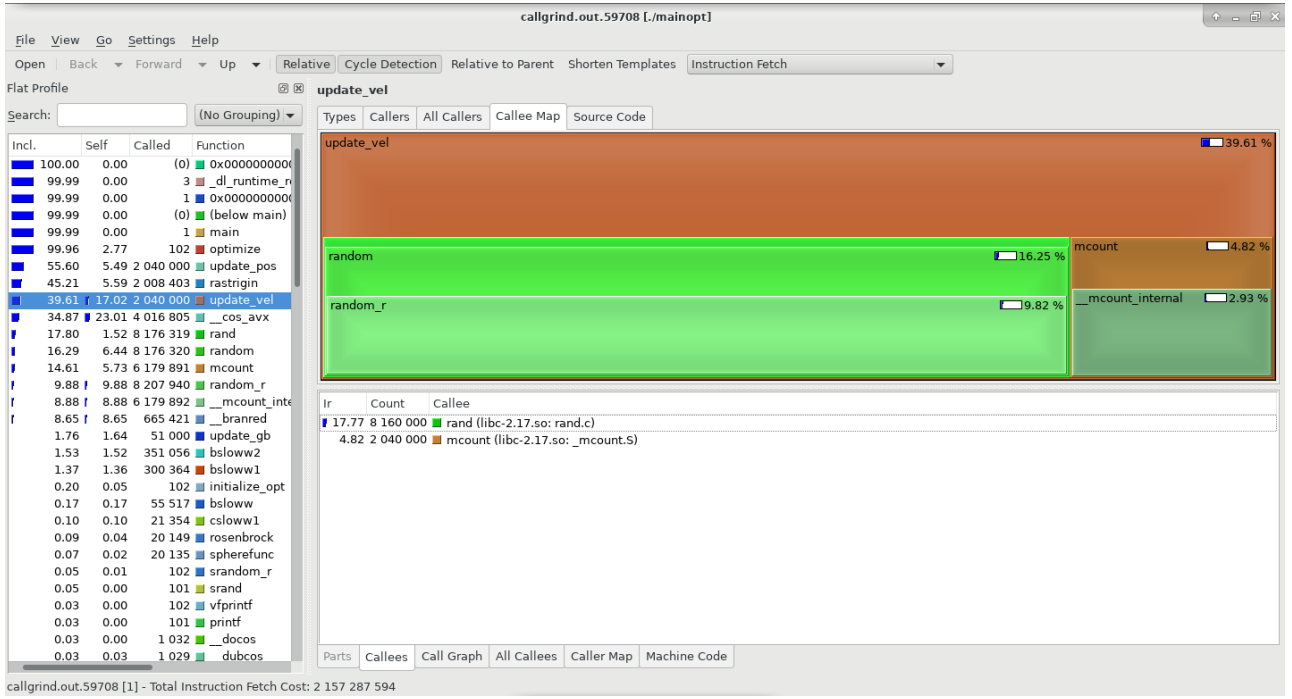
$valgrind –tool=callgrind

```
$valgrind --tool=callgrind ./mainopt
```

Creates `callgrind.out.X`
You can read output file

But its tricky; try:
```
$kcachegrind callgrind.out.X
```

(Must install cachegrind / needs graphical interface)

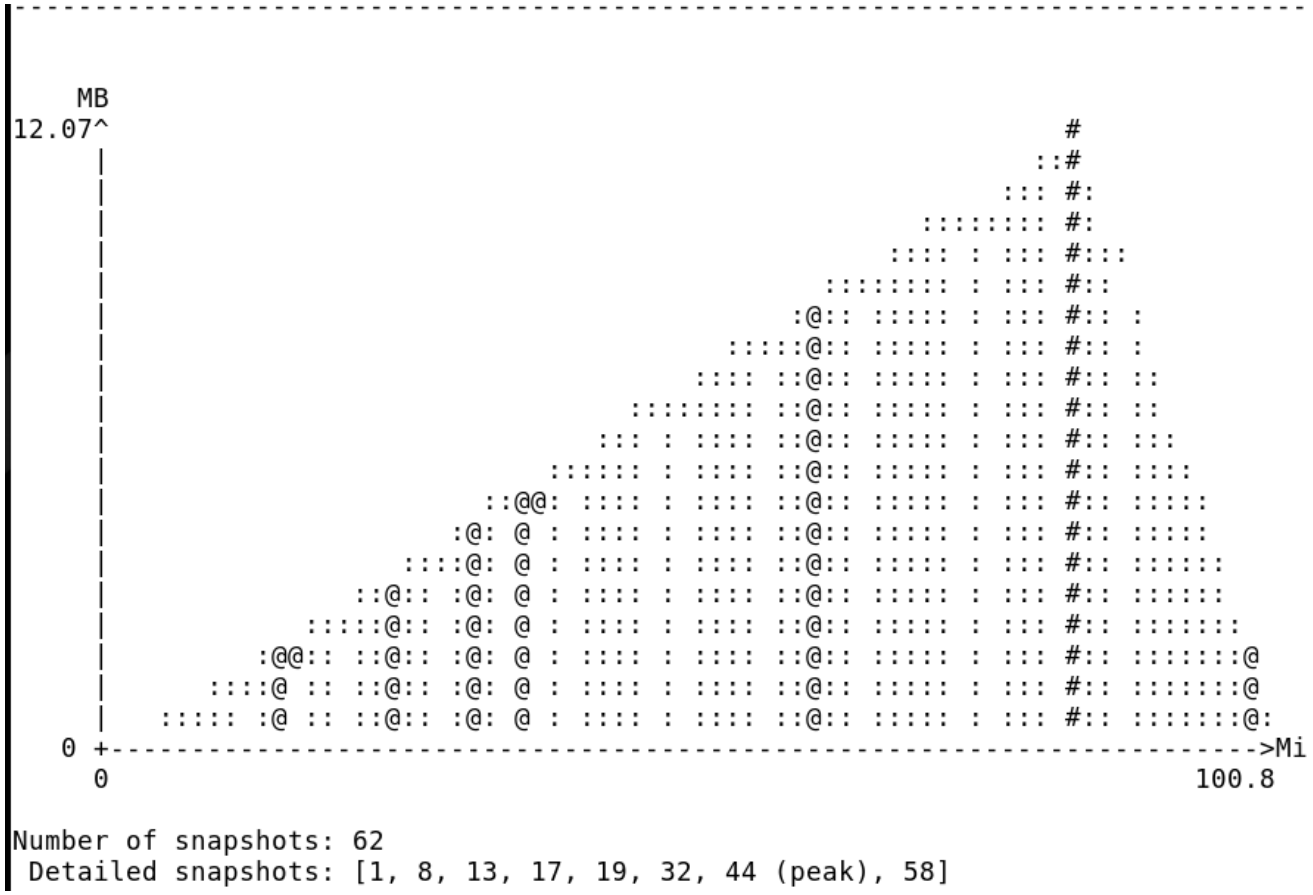


$valgrind –tool=massif

`$valgrind --tool=massif ./mainopt`

Creates `massif.out.X`
You can read output file

But its tricky; try:
`$ms_print massif.out.X`



Demo: Search?

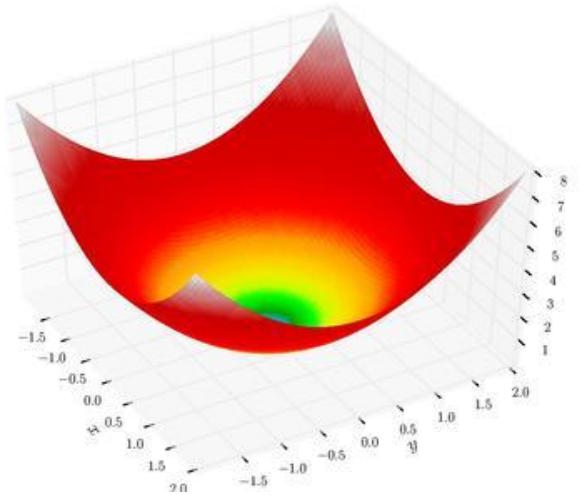
Search has a large number of applications and shows up in many different fields:

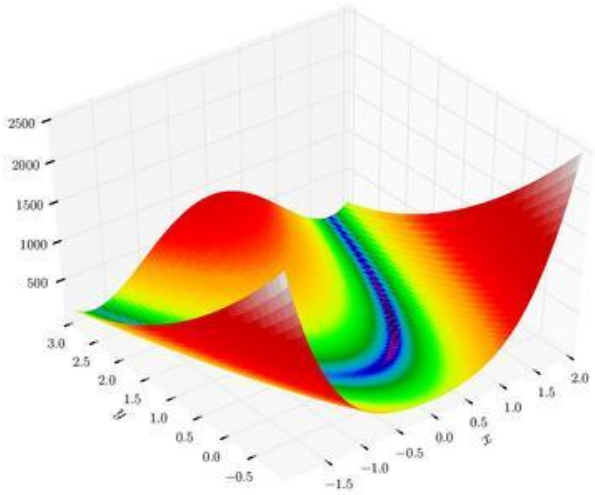
- Optimize control settings (intelligent control)
- Find low energy states (protein structure determination)
- Fit functions to data (machine learning)
- Train deep neural networks (LLMs)

- Minimize a function $f0 : R^n \rightarrow R$, the objective function, over a specified set $C \subset R^n$, the feasible set
- Feasibility: $x \in C$
- Optimality: a feasible solution such that $f0(x^*) \leq f0(x)$ for all other feasible solutions x

Demo search benchmarks

- Benchmark functions represent different complexities, often with different levels of complexity.
  - Sphere function nicely behaved convex function
  - Rosenbrock quali-convex with poorly behaved slope
  - Rastrigin (not pictured) non-convex and highly multi-modal
- Can you solve the benchmark?
  - How close do you get?
- How long does it take?
- How much memory?



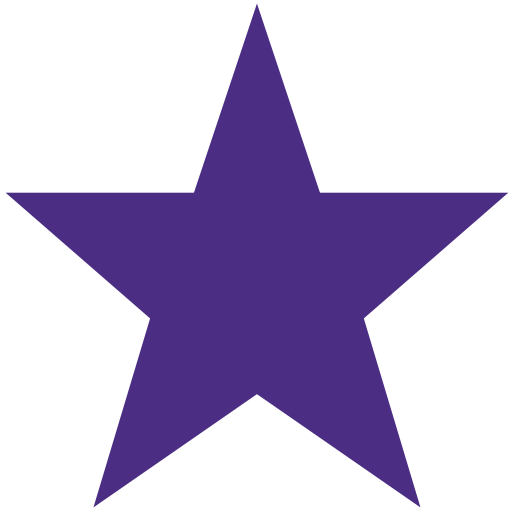


Demo: Particle Swarm

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling. o Akin to genetic algorithms…

Used to find the global optimum of potentially non-convex functions.

```
float* optimize(float (*obj)(float*), float* mins,
  float* maxs) {
    // initialize population
  particle* population;
  // global best position and global best value
  float* gb =(float*) malloc(DIM*sizeof(float));
  float gbval;
  population = initialize_opt(obj, mins, maxs, gb,
  &gbval);
  // for MAXITS update population and global best
  for (int i = 0; i < MAXITS; i++) {
    update_gb(population, gb, &gbval);
    for (int p = 0; p < POPSIZE; p++) {
        update_vel(&population[p], gb);
        update_pos(obj, &population[p]);
    }
  }
  free(population);
  return gb;
}
```



EXTRAS

Memory Model

Code Globals Heap-> <-Stack

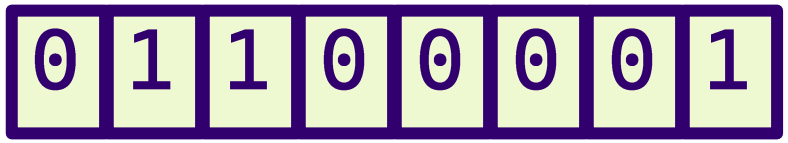
- Each byte of memory has an address
- Lowest memory stores program instructions, then global variables (static constants, string literals)
- 'Heap' holds dynamically allocated variables ('new' or 'malloc' variables)
- 'Stack' holds current instructions, each function in a frame
  - 'Stack' memory implies that a frame is added, and then the last frame added is removed first

Memory (zoomed in) (bkup)

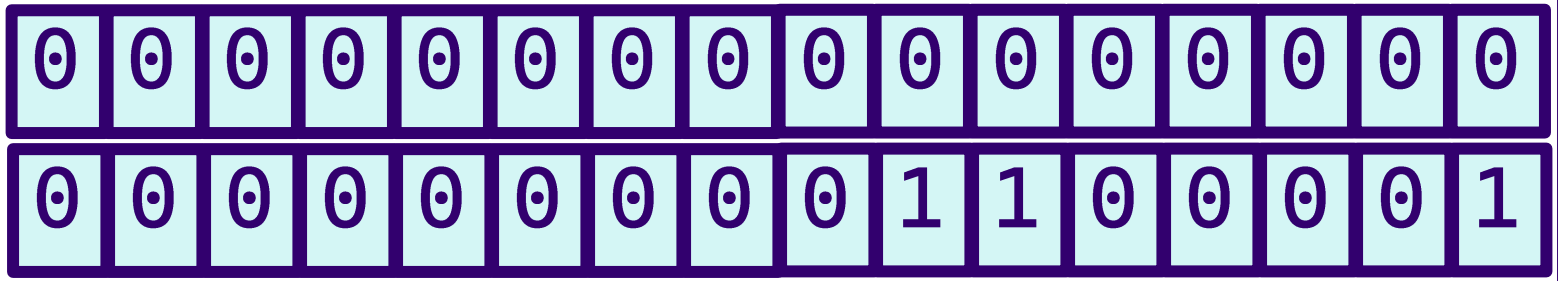
char – 1 byte **int – 4 bytes**

- Each byte of memory has an address
- Different data types need different amount of memory to store them
- All data types are stored as binary code: each of the 8 bits in a byte is a zero or one.

Memory (zoomed in even more)



char – 'a' in binary



int – 97 in binary

- All data types are stored as binary code: each of the 8 bits in a byte is a zero or one.