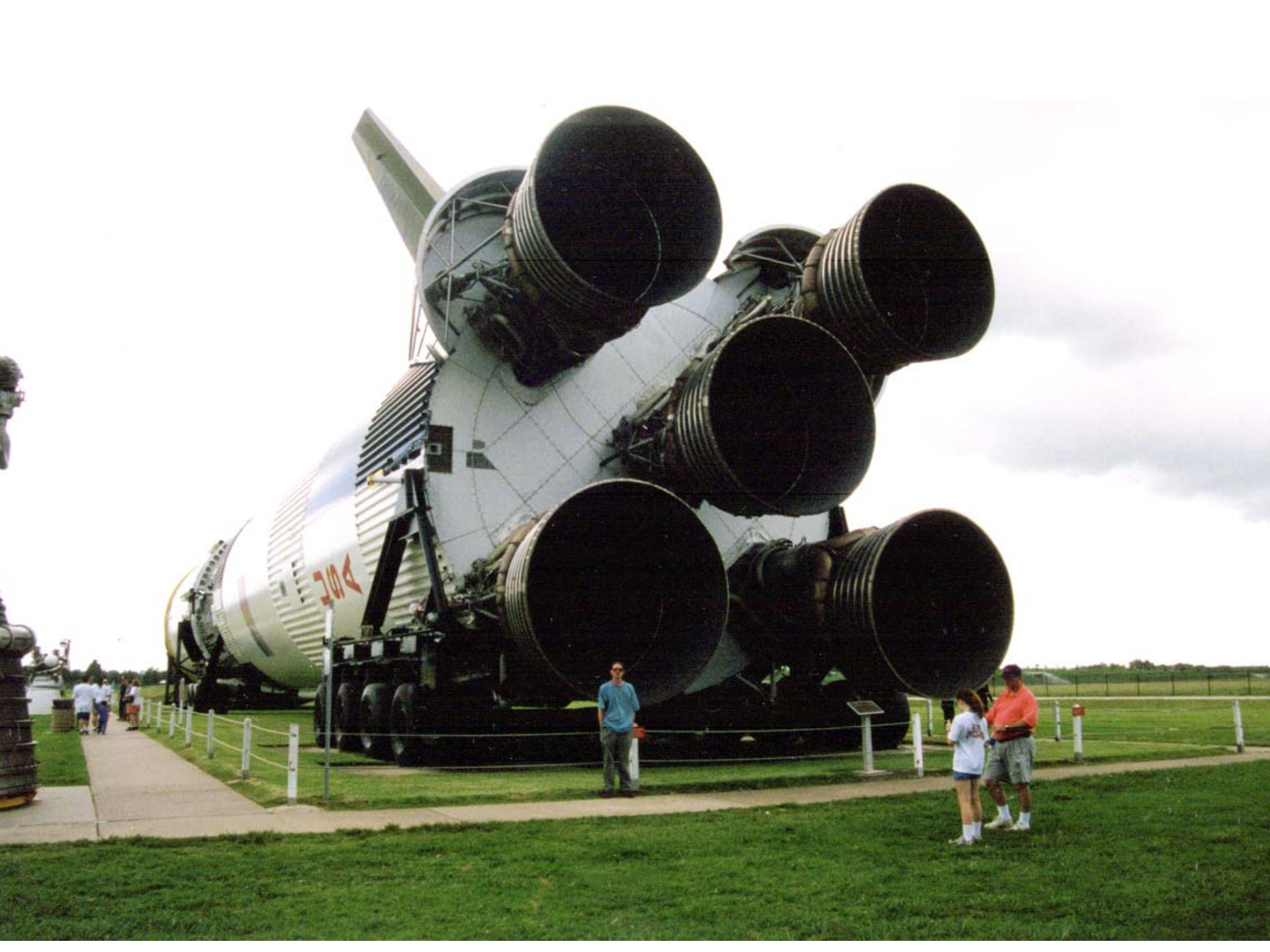
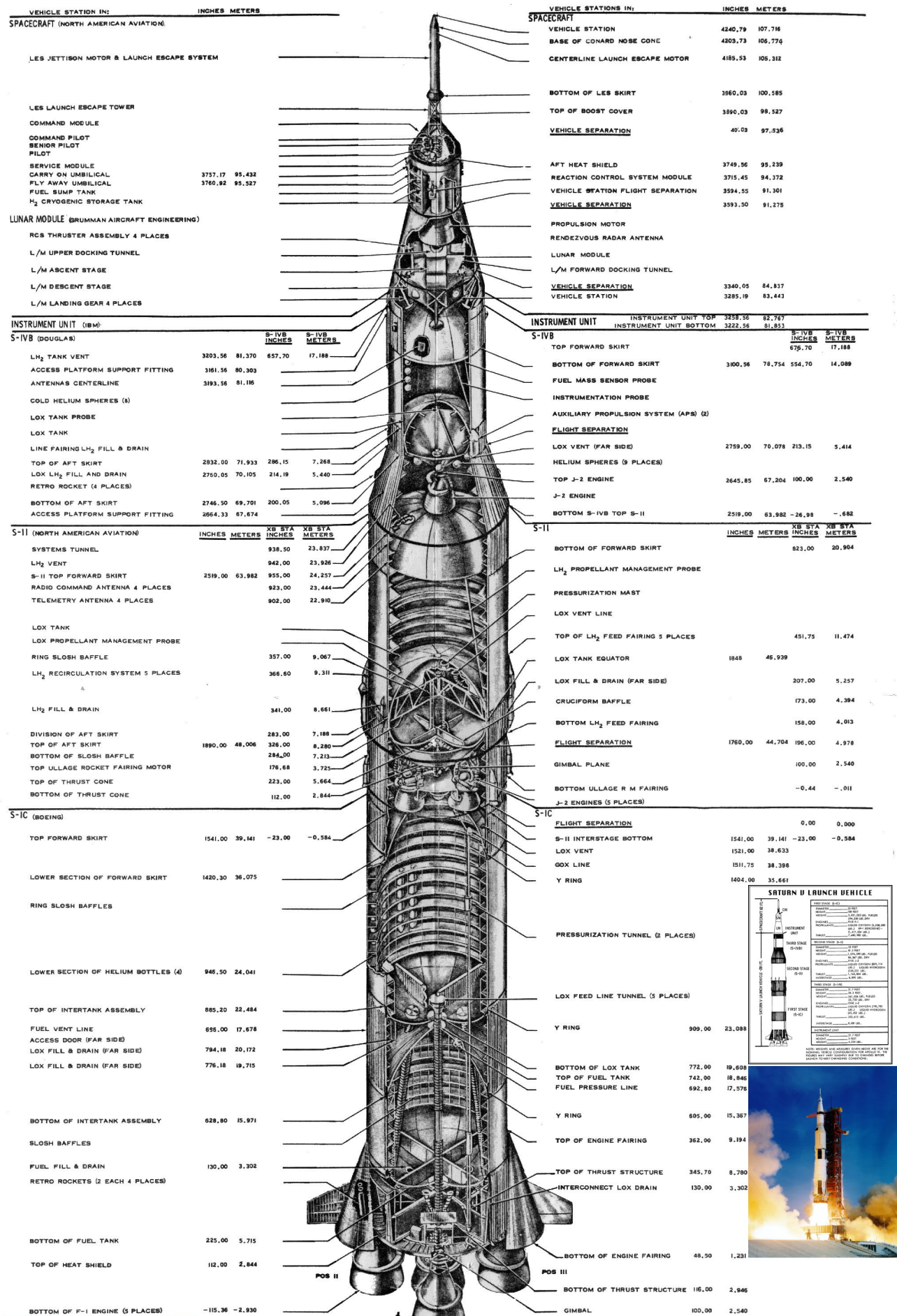


Performance

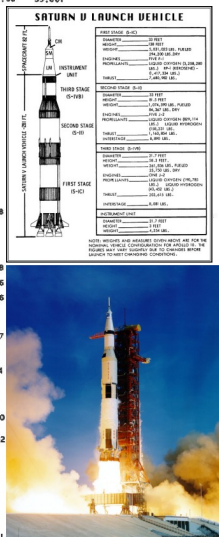




Command Module
64,000 lbs

Saturn V
6,200,000 lbs

Payload
1.5% of total weight



Falcon 9

Elon Musk





Demo: optimize gl_clear

Optimization Strategies

- Inlining - Reduce function call overhead
- Avoid volatile - enable more compiler optimizations
- Load and Store Larger Chunks of Data - use less instructions per memory operation
- Loop unrolling - Reduce loop overhead
- Inline assembly - Be smarter than the compiler

Amdahl's Law

$$S_{latency}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

- $S_{latency}$ - the theoretical speedup of the execution of the entire program
- s - the speedup of the part of the program you're optimizing
- p - the proportion of execution time that the part the program you're optimizing originally occupied

Amdahl's Law



Suppose your original program took t cycles to execute

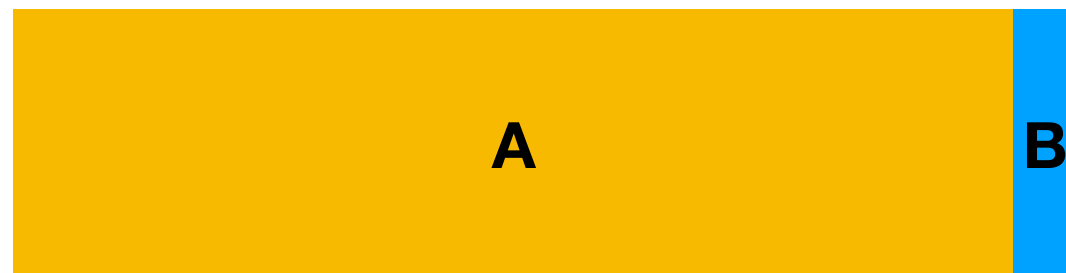
Amdahl's Law



The program is divided into two distinct portions:

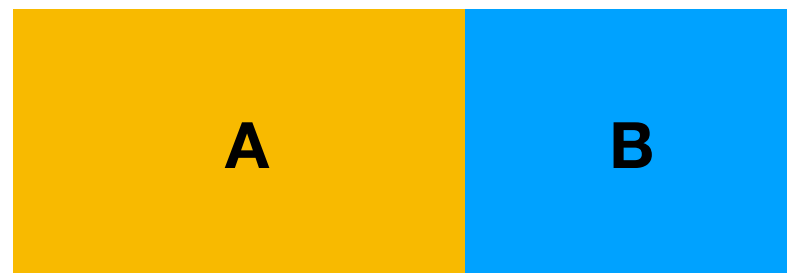
- Part **A** takes 75% of the time
- Part **B** takes 25% of the time

Amdahl's Law



If we optimize part B to make it 5 times faster, this only reduces the overall computation time slightly

Amdahl's Law



If we optimize part A to make it just twice as fast, we get a greater overall speedup

Amdahl's Law

Part B is 25% of the overall program ($p = .25$) and we speed it up by a factor of 5 ($s = 5$)

$$S_{latency} = \frac{1}{1 - .25 + \frac{.25}{5}} = 1.25$$

Overall program speedup is 1.25

Amdahl's Law

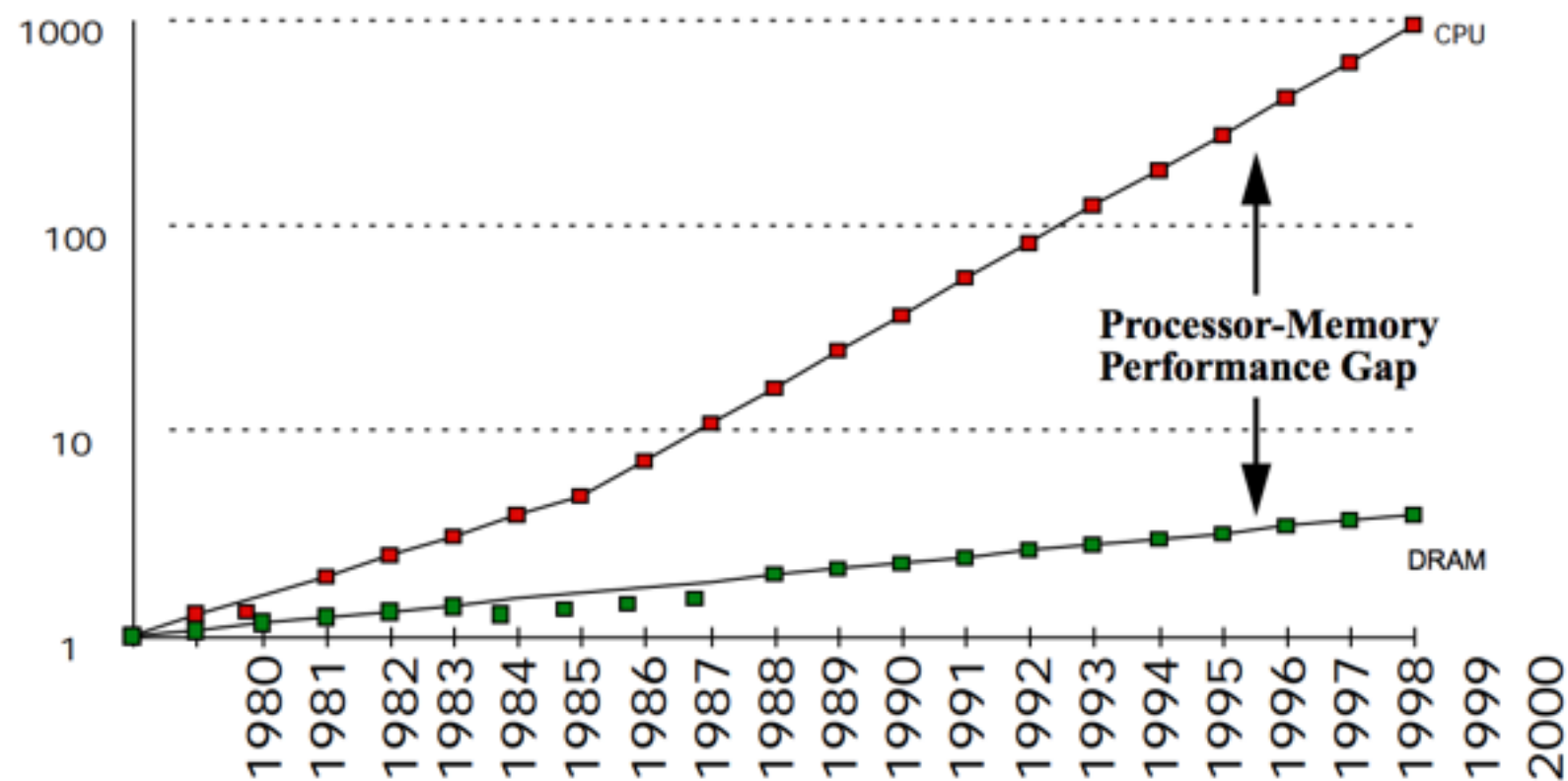
Part A is 75% of the overall program ($p = .75$) and we speed it up by a factor of 2 ($s = 2$)

$$S_{latency} = \frac{1}{1 - .75 + \frac{.75}{2}} = 1.60$$

Overall program speedup is 1.60

Beyond Software Optimization

Processor-Memory Performance Gap



[Hennessy, J.L.; Patterson, D.A. Computer Organization and Design, 2nd ed.]

Memory System Performance

Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)	960 MB/sec
Main Memory Subsystem	253 ns (76 clocks)	1200 MB/sec
Single DRAM component	≈60ns (18 clocks)	≈30–100 MB/sec

[Patterson, David, et al. "A case for intelligent RAM."]

Moving data between the
cpu and memory is the
bottleneck

strcpy

```
for (int i = 0; i <= strlen(src); i++) {  
    dst[i] = src[i]  
}
```

All we're doing is loading data from memory into the CPU and storing it back into memory

Avoiding the Memory Bottleneck

- Raspberry Pi has a **DMA Controller** that allows us to read and write memory without having to go through the processor (avoiding the load/store latency)
- Section 4 of BCM2835-ARM-Peripherals.pdf

Demo: dma_strcpy.c

Measuring Performance

Don't optimize blind

Profiling

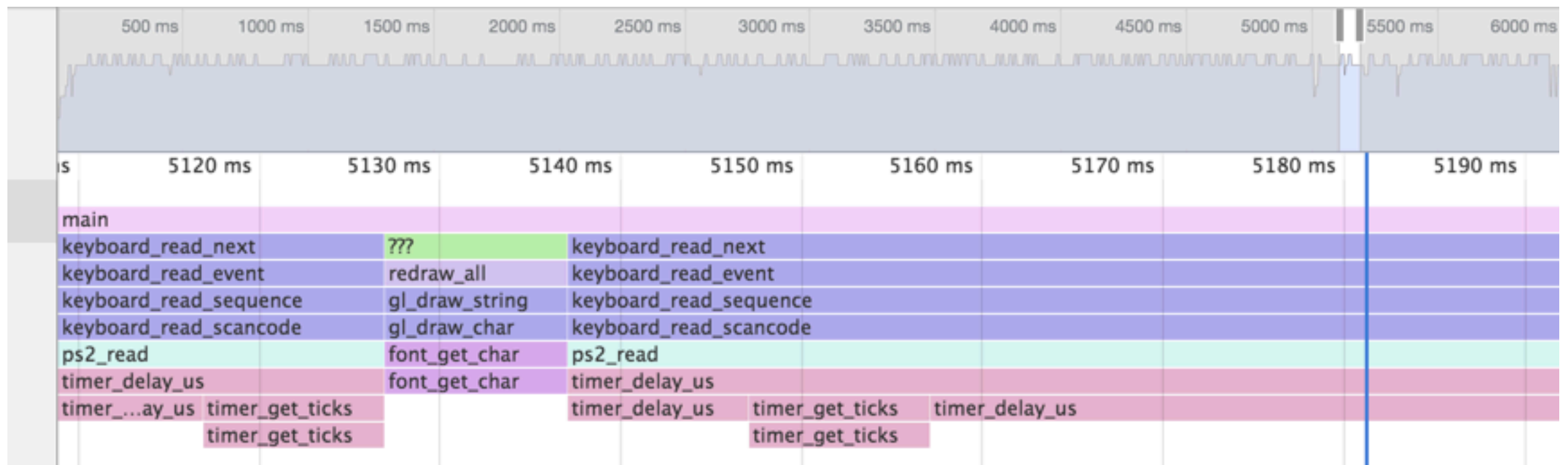
- Analyze your program at runtime to measure characteristics of interest
 - Space/time complexity
 - Frequency of certain instructions
 - Frequency and Duration of Function Calls
- Most often used for guiding optimization

Taking Measurements

- **Hardware interrupts** - gprof.c
- **Code Instrumentation** - timer
- Also: instruction set simulation, OS hooks, performance counters
- Many techniques rely on sampling (statistical profilers) to trade off accuracy for speed

Visualizing Measurements

Chrome Developer Tools

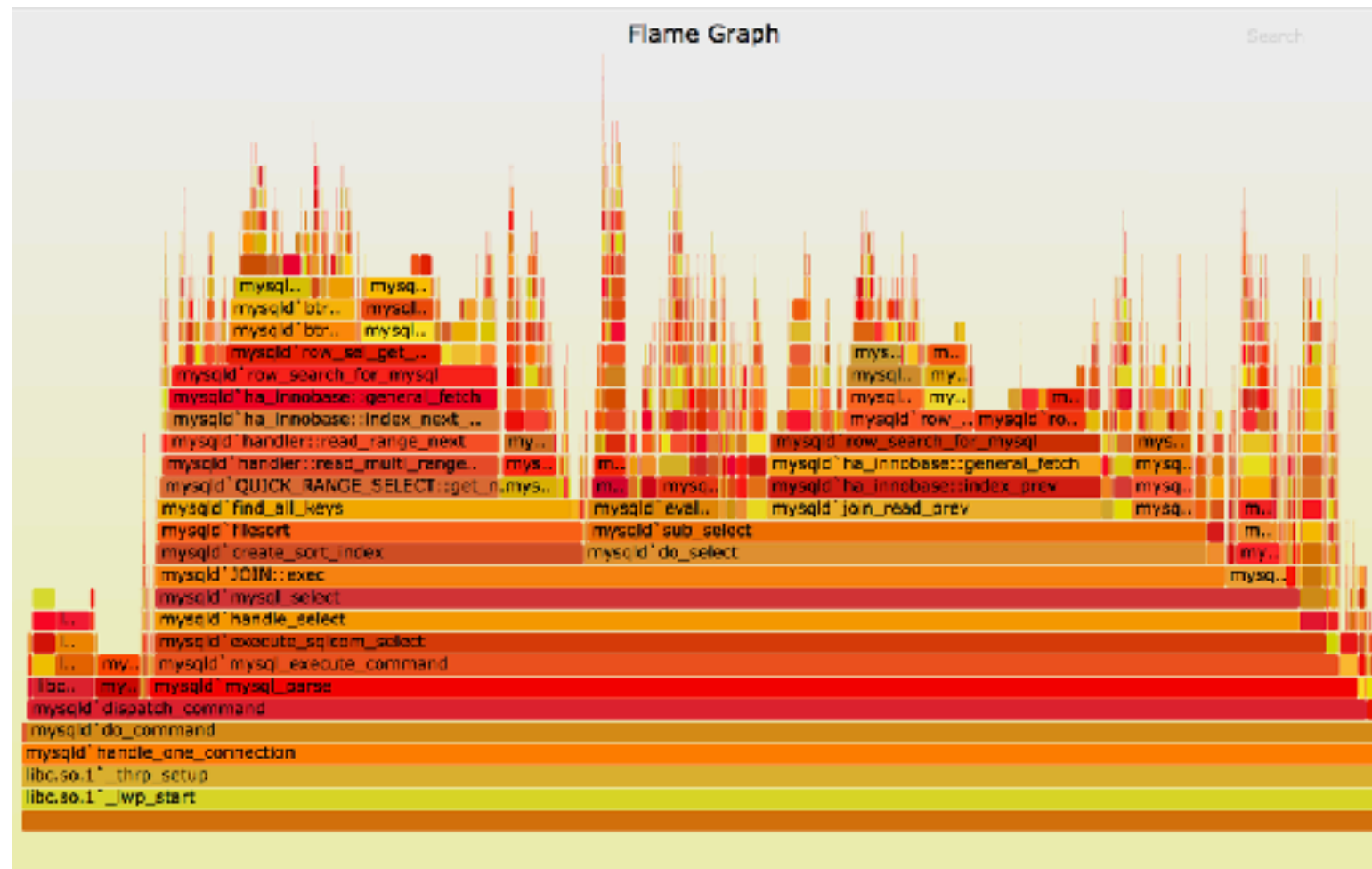


- Output profiling information in standard format (linux *perf*)
- Use **thlorenz/cpuprofilify** to convert into *.cpuprofile* format

Demo: `stackprof.c`

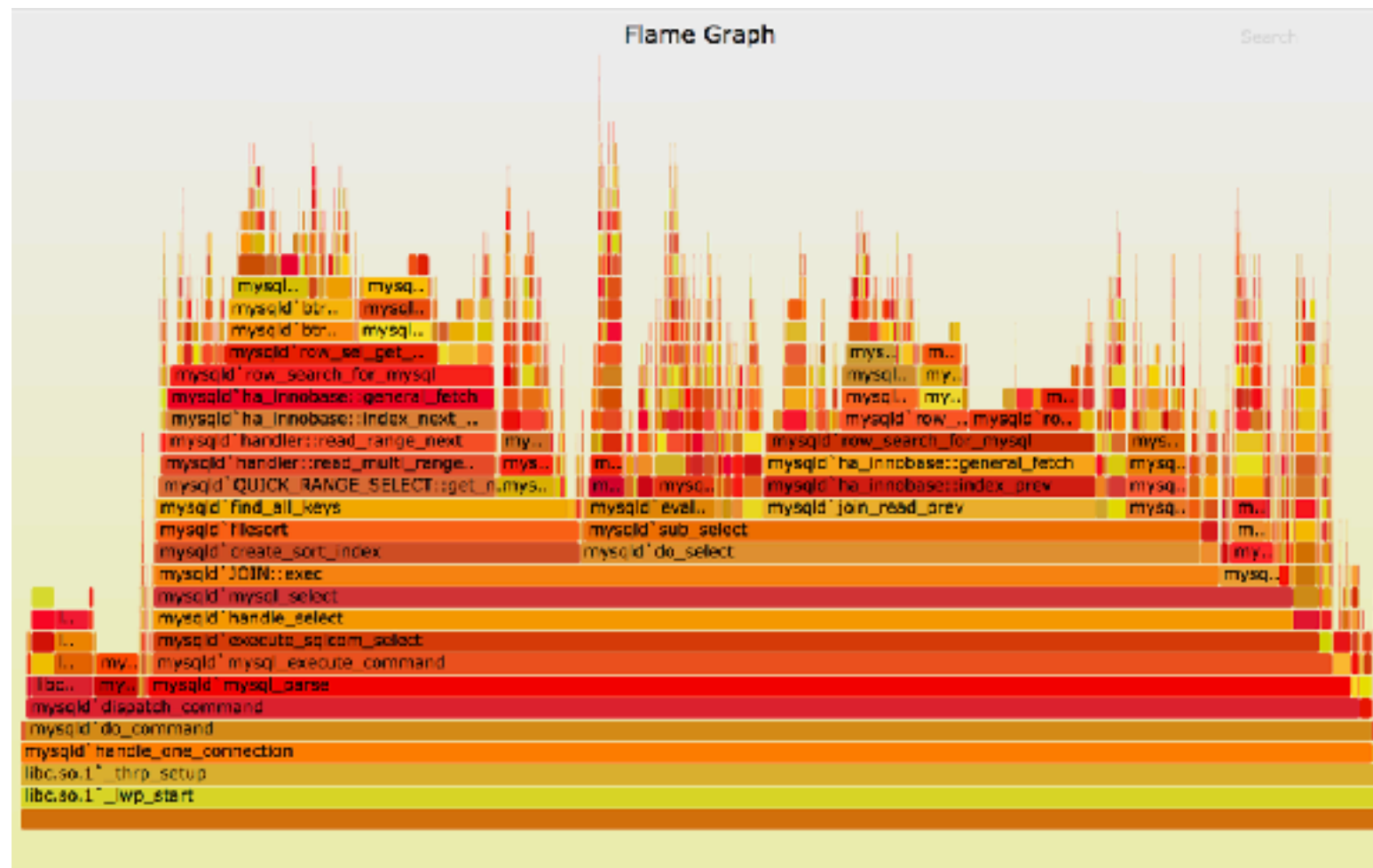
Visualizing Measurements

Flame Graph - designed to quickly and accurately find the most frequent code paths



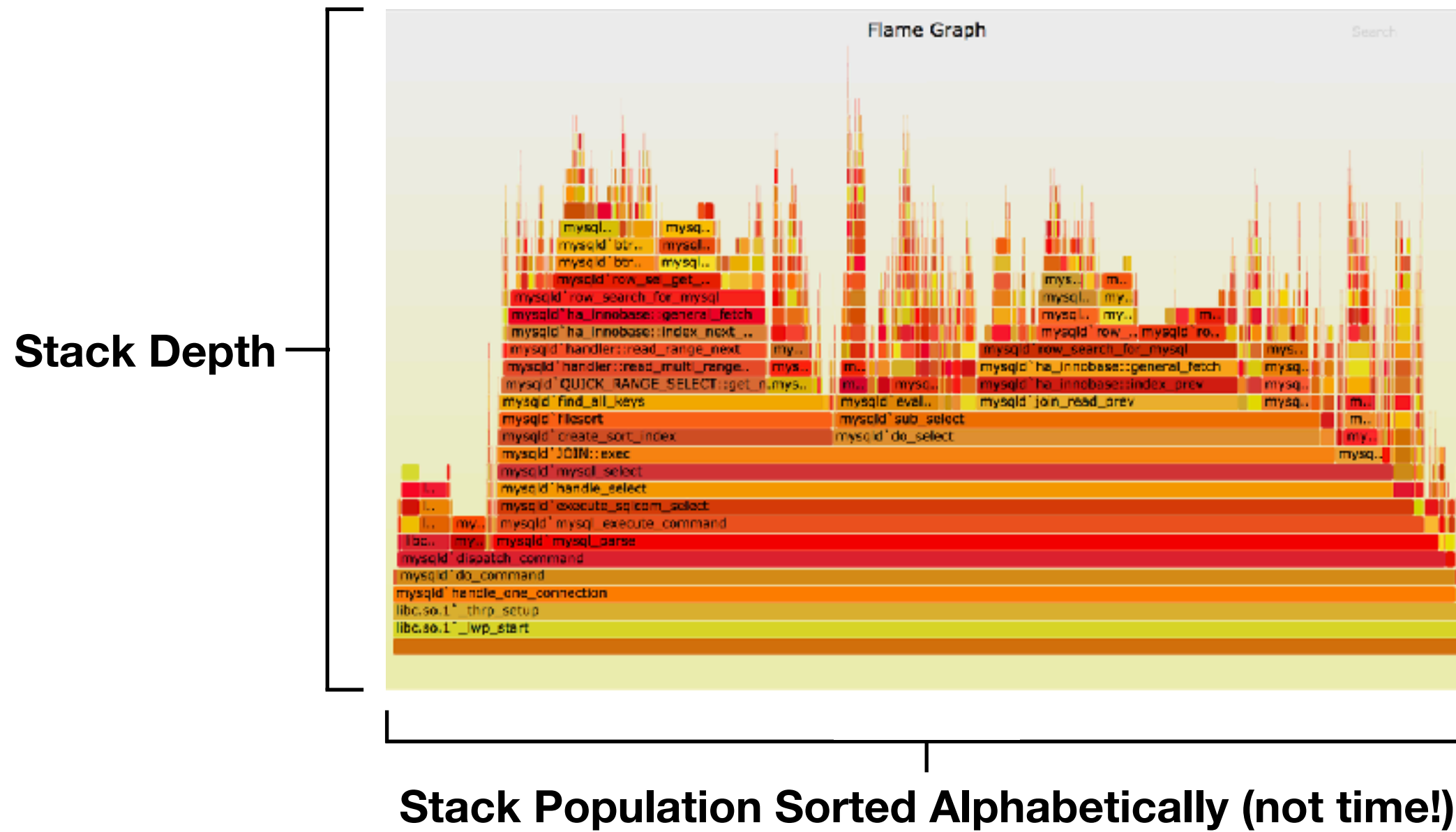
<http://www.brendangregg.com/flamegraphs.html>

Flame Graph



Stack Population Sorted Alphabetically (not time!)

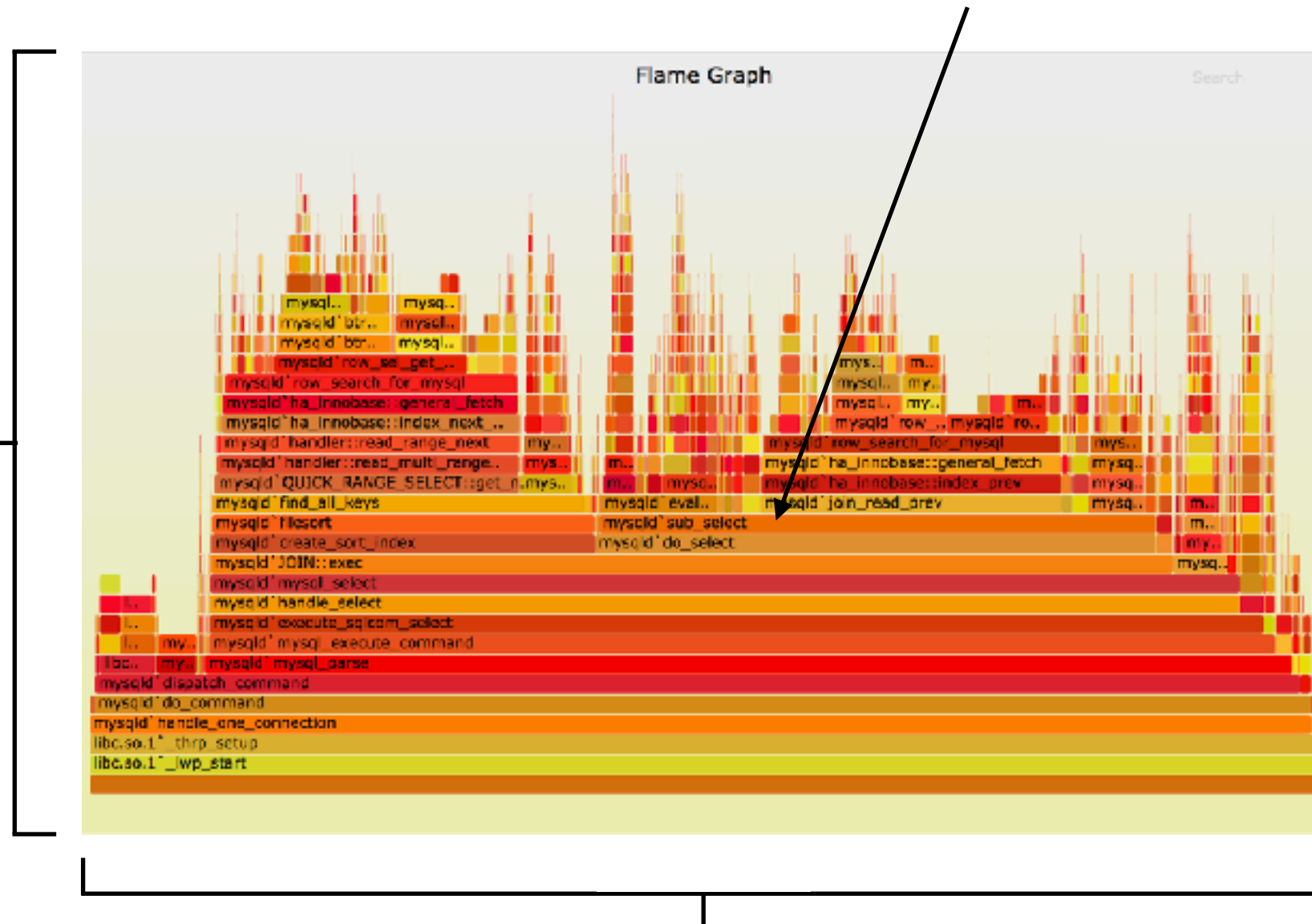
Flame Graph



Flame Graph

Rectangle is a stack frame

Stack Depth



Flame Graph

Wider the frame, the more often is was on present in the stack

