

Performance Instrumentation

Lecture 08

Experimentation Challenges

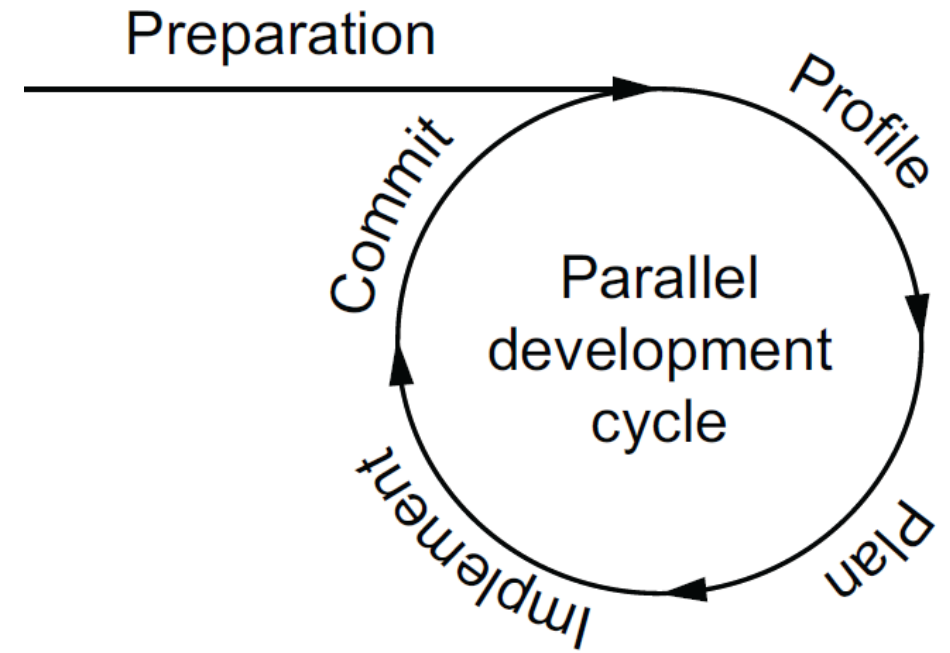
- Experiment with writing and tuning your own parallel programs
 - Many times, we obtain misleading results or tune code for a workload that is not representative of real-world use cases
- Start by setting your application performance goals
 - Response time, speedup?
 - Throughput, utilization?
 - Determine if your evaluation approach is consistent with these goals

Aspects

- Measures the quality attributes of the system
 - ❑ Scalability
 - ❑ Reliability
 - ❑ Resource usage
- Workload: normal (loaded) and overloaded
- Timeline
 - ❑ Not time sensitive: testing before release
 - ❑ Time sensitive: incident performance response (Software Reliability Engineering - SRE)

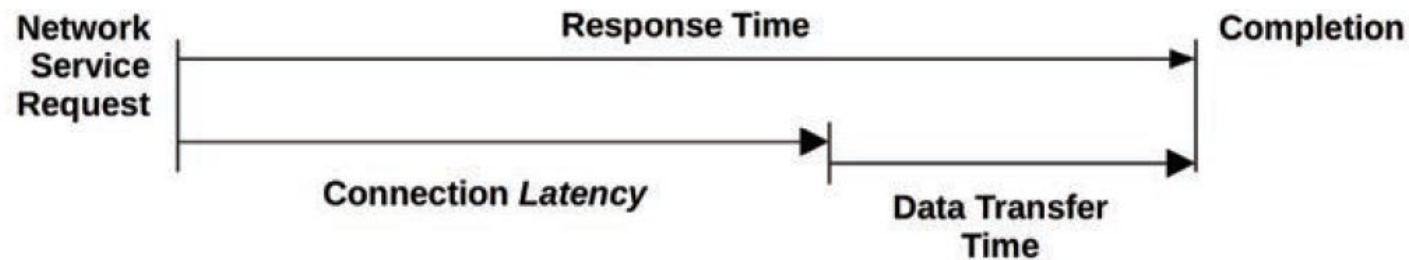
Overview of Performance Instrumentation

- Our approach for analyzing the performance of programs
 - Analyze performance bottlenecks
 - Tune codes to improve timing in normal/stressing scenarios



Terminology (1)

- **Workload**: The input to the system or the load applied is the workload. For a database, the workload consists of the database queries and commands sent by the clients.
- **Latency**: A measure of time an operation spends waiting to be serviced. In some contexts, it can refer to the entire time for an operation, equivalent to response time
- **Response time**: The time for an operation to complete. This includes any time spent waiting and time spent being serviced (service time), including the time to transfer the result.

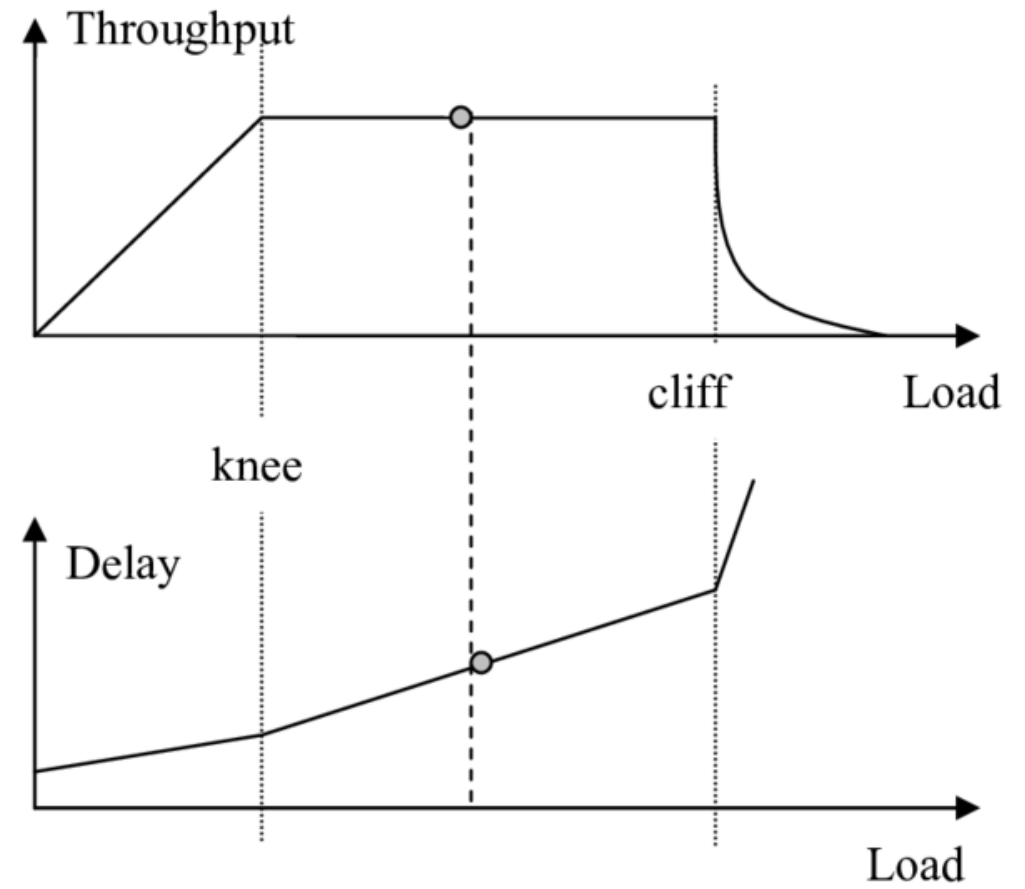


- **Throughput**: The rate of work performed. Especially in communications, the term is used to refer to the data rate (bytes per second or bits per second). In some contexts (e.g., databases) throughput can refer to the operation rate (operations per second or transactions per second).

Terminology (2)

- **Utilization**: For resources that service requests, utilization is a measure of how busy a resource is, based on how much time in a given interval it was actively performing work. For resources that provide storage, utilization may refer to the capacity that is consumed (e.g., memory utilization).
- **Saturation**: The degree to which a resource has queued work it cannot service.
- **Bottleneck**: In systems performance, a bottleneck is a resource that limits the performance of the system. Identifying and removing systemic bottlenecks is a key activity of systems performance.

Knee point



Understanding System's Performance

- Perspectives

- Resources
- Workload

- Methodologies

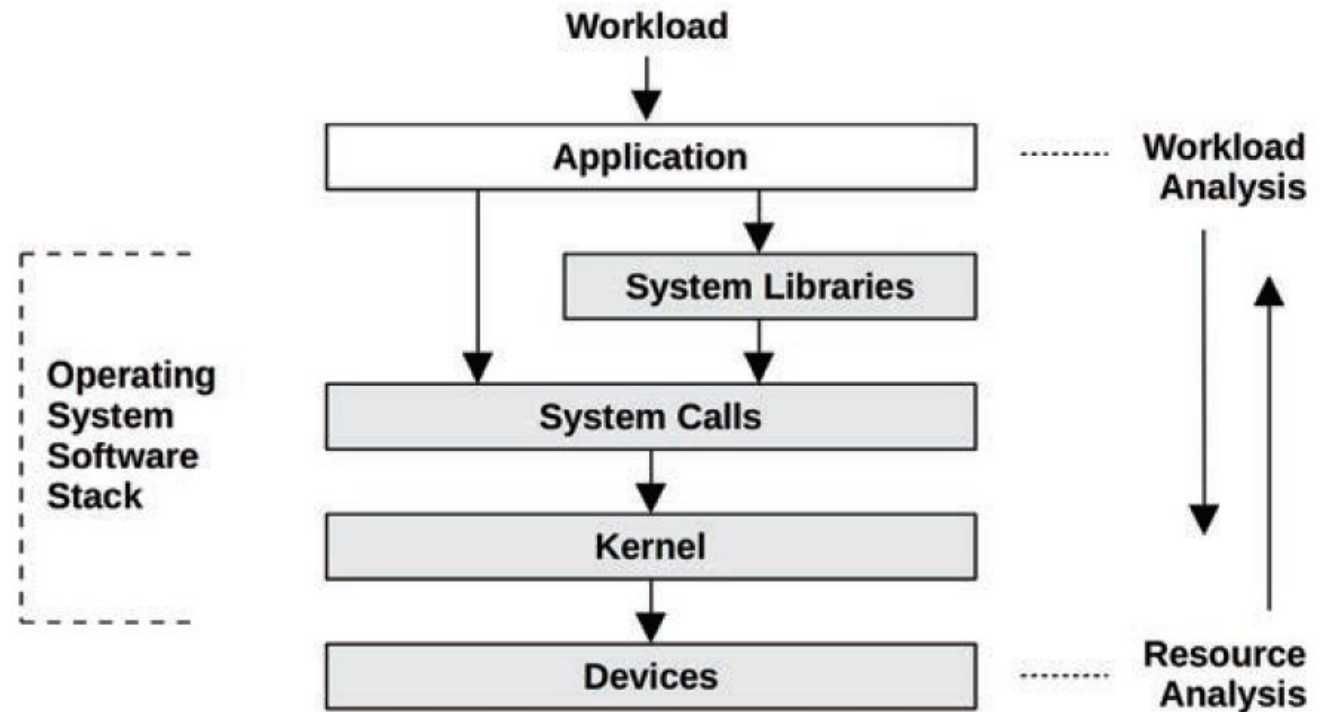
- Observability Tools

- Counters and Profiling
- Debugging Tools

- Benchmarking

Perspectives

- The resource analysis perspective → system administrators
- The workload analysis perspective → application developers



Resource analysis

- Begins with analysis of the system resources: CPUs, memory, disks, network interfaces, buses, and interconnects
- Includes
 - Performance issue investigations
 - Capacity planning
- Focusses on **utilization**
 - Demand - supply

Workload analysis

- Examines the workload applied and how the application is responding
- Targets
 - Requests: The workload applied
 - Latency: The response time of the application
 - Completion: Looking for errors
- Metrics
 - **Throughput**
 - **Latency**

Methodologies

- Know where to begin analysis and how to proceed
- Anti-methods
 - ❑ Random change anti-method
 - ❑ Streetlight anti-method
 - ❑ Blame-someone-else anti-method
- Methods
 - ❑ Problem statement method
 - ❑ USE method
 - ❑ Performance monitoring
 - ❑ Performance analysis in 60 mins
 - ❑ Tools method
 - ❑ CPU profile method

Anti-methodologies

- The lack of a deliberate methodology...
 - Tune things too early
- Street Light Anti-Method
 - Look for obvious issues in tools that are
 - Familiar
 - Found on the Internet
 - Found at random
- Drunk Man Anti-Method
 - Tune things at random until the problem goes away
 - Tune the wrong software (OS instead of application)

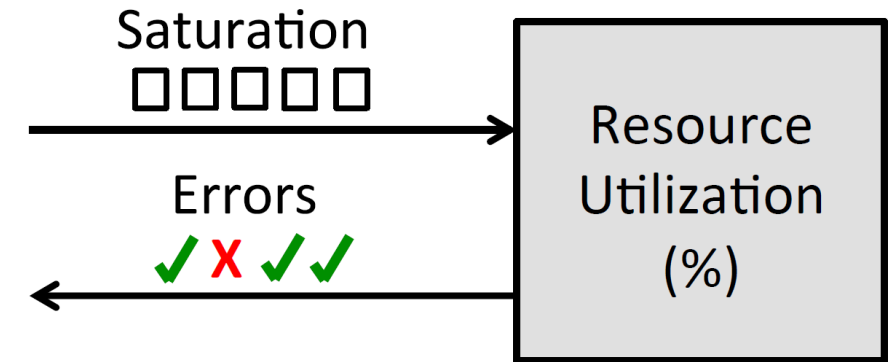
Problem Statement Method

1. What makes you think there is a performance problem?
2. Has this system ever performed well?
3. What has changed recently? (Software? Hardware? Load?)
4. Can the performance degradation be expressed in terms of latency or run time?
5. Does the problem affect other people or applications (or is it just you)?
6. What is the environment? Software, hardware, instance types? Versions? Configuration?

USE Method

- For every resource (CPU, memory, bus, etc.), check

- **Utilization:** busy time
- **Saturation:** queue length or queued time
- **Errors:** easy to interpret (objective)



- Helps if you have a functional (block) diagram of your system / software / environment, showing all resources
- Start with the questions, then find the tools

Monitoring

- Records performance statistics over time (a time series) so that the past can be compared to the present and time-based usage patterns can be identified
- Useful for
 - Capacity planning
 - Quantifying growth
 - Showing peak usage
- Historic values can also provide context for understanding the current value of performance metrics
 - Time-based patterns

Performance Analysis in 60 Sec (Linux)

- | | | |
|-------------------------|--------|----------------------|
| ■ Load averages | —————→ | 1. uptime |
| ■ Kernel errors | —————→ | 2. dmesg tail |
| ■ Overall stats by time | —————→ | 3. vmstat 1 |
| ■ CPU balance | —————→ | 4. mpstat -P ALL 1 |
| ■ Process usage | —————→ | 5. pidstat 1 |
| ■ Disk I/O | —————→ | 6. iostat -xz 1 |
| ■ Memory usage | —————→ | 7. free -m |
| ■ Network I/O | —————→ | 8. sar -n DEV 1 |
| ■ TCP stats | —————→ | 9. sar -n TCP,ETCP 1 |
| ■ Check overview | —————→ | 10. top |

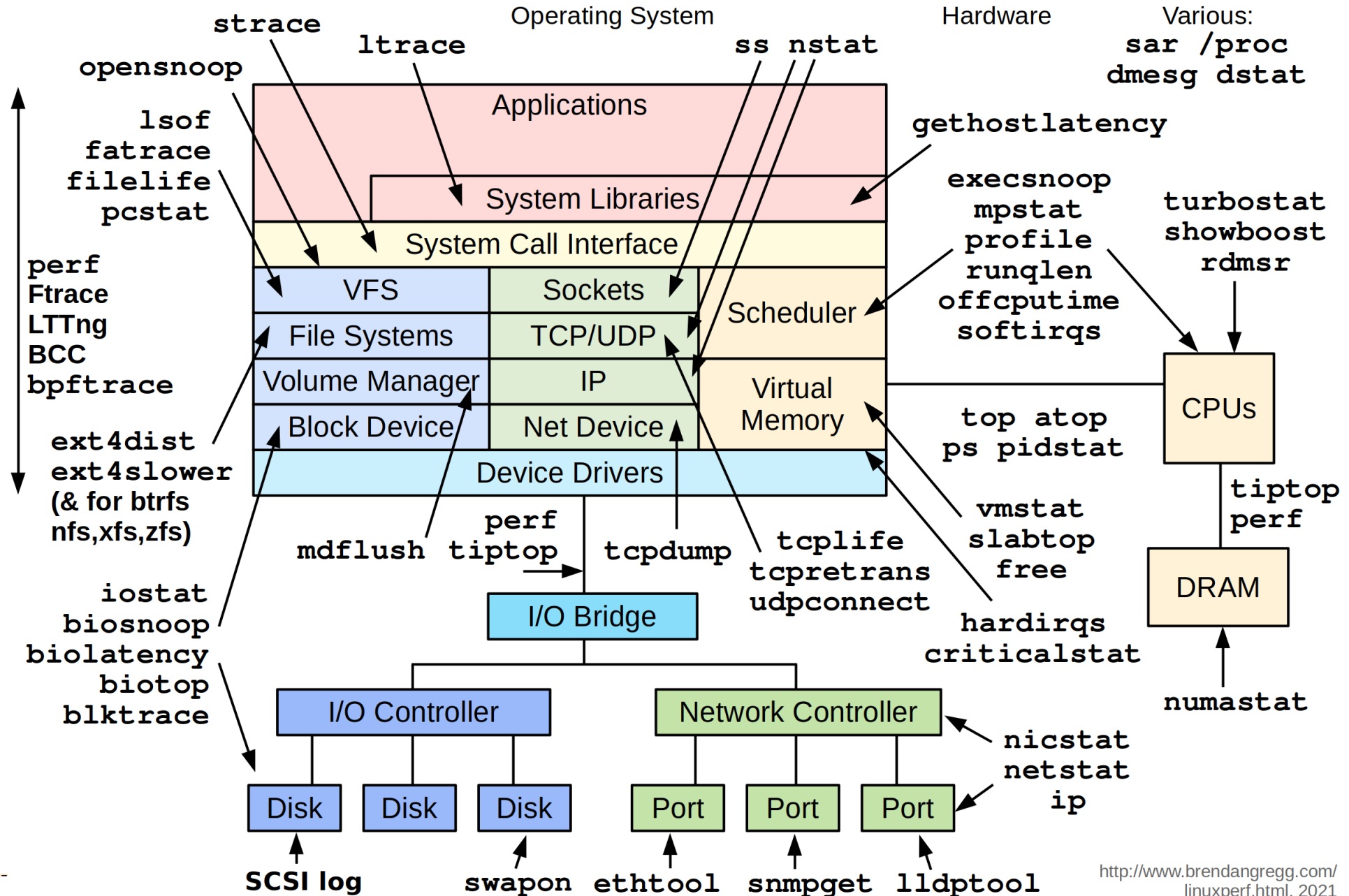
Tools method

1. List available performance tools (optionally, install or purchase more).
2. For each tool, list useful metrics it provides.
3. For each metric, list possible ways to interpret it.

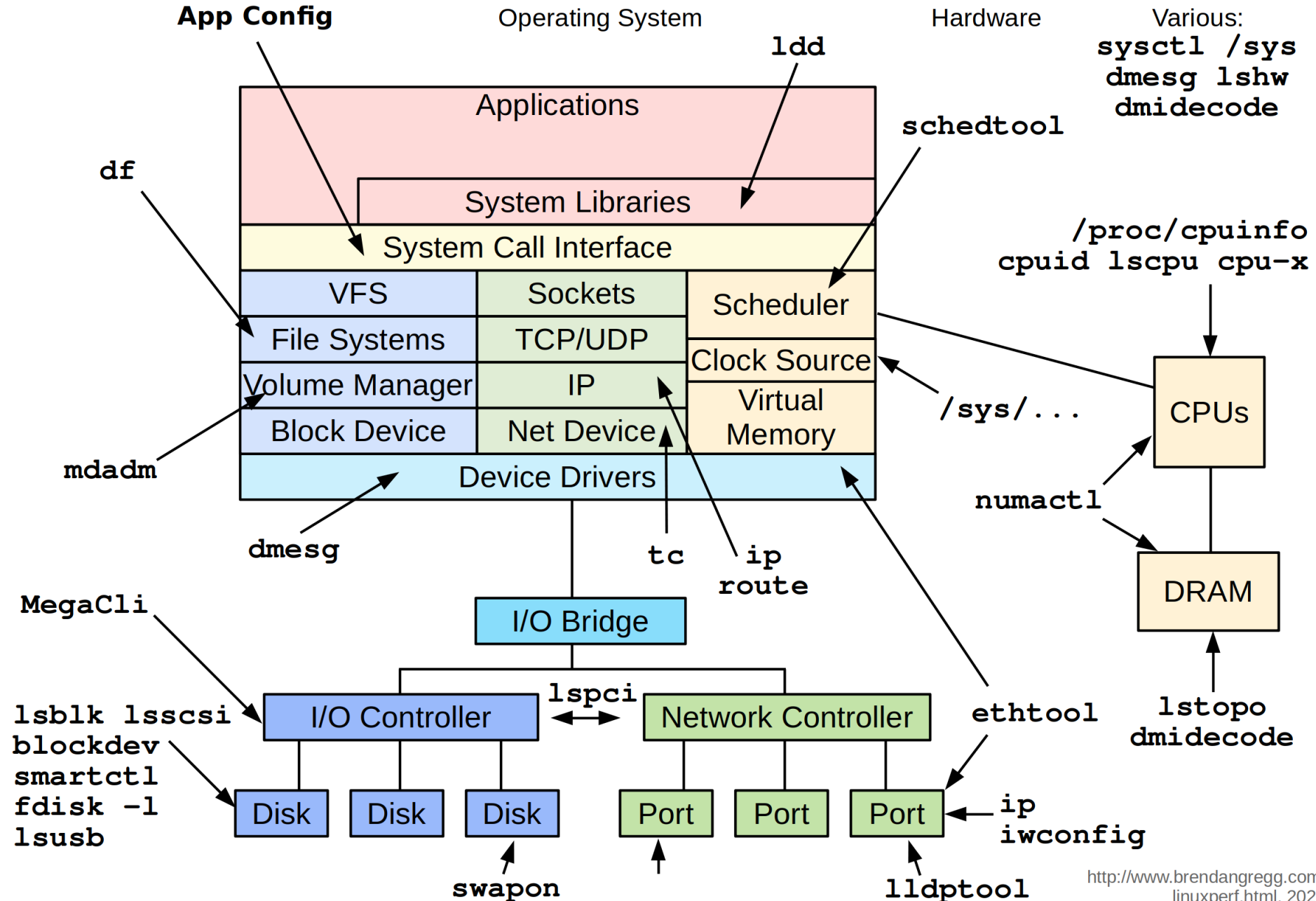
Types of Tools

Type	Characteristic
Observability	Watch activity under workload. Safe, usually, depending on resource overhead. <ul style="list-style-type: none">-insert timing statements-check performance counters
Static	Examine attributes of the system at rest rather than under active workload. Should be safe.
Benchmarking	Load test. Caution: production tests can cause issues due to contention.
Tuning	Change default settings. Danger: changes could hurt performance, now or later with load.

Linux Performance Observability Tools

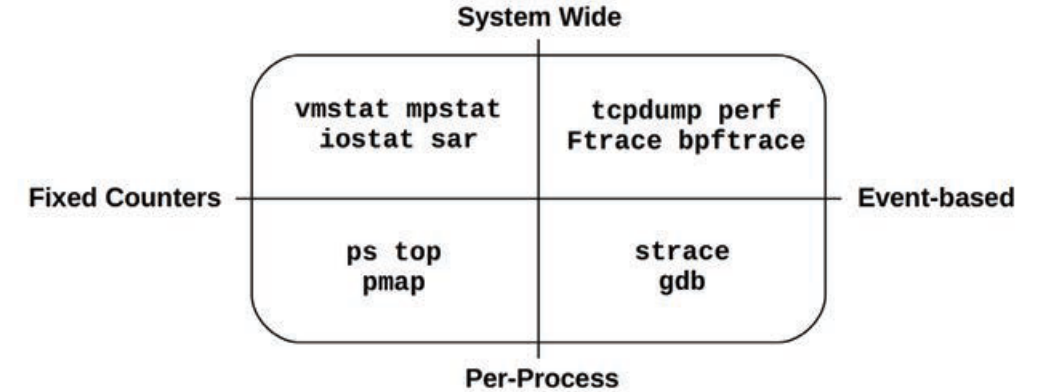


Linux Static Performance Tools



Tools Categorization

- System-wide or per-process observations
- Fixed counters:
 - ❑ Kernels maintain various counters for providing system statistics
 - ❑ Kernels expose these **counters** as metrics
- Event-based counters:
 - ❑ Enabled as needed
 - ❑ **Profiling**: characterizes the target by collecting a set of samples or snapshots of its behavior
 - ❑ **Tracing**: instruments every occurrence of an event, and can store event-based details for later analysis or produce a summary



Instrumentation Tools

- Modify the source code, executable or runtime environment to understand the performance
 - ❑ Manual: Performed by the programmer
 - ❑ Automatic source level: instrumentation added to the source code by an automatic tool
 - ❑ Intermediate language/Compiler assisted: instrumentation added to assembly or decompiled bytecodes
 - ❑ Binary translation: adds instrumentation to a compiled executable.
 - ❑ Runtime instrumentation: program run is fully supervised and controlled by the tool.
 - ❑ Runtime injection: code is modified at runtime to have jumps to helper functions.

Overhead

- The execution is slowed down when instrumentation is added
- Ideally, minimal impact is desired
- Time measurements might not be accurate, but other measurements should be useful
 - Finding bottlenecks
 - Debugging

Methodologies

- Know where to begin analysis and how to proceed
- Anti-methods
 - ❑ Random change anti-method
 - ❑ Streetlight anti-method
 - ❑ Blame-someone-else anti-method
- Methods
 - ❑ Problem statement method
 - ❑ USE method
 - ❑ Performance monitoring
 - ❑ Performance analysis in 60 mins
 - ❑ Tools method
 - ❑ CPU profile method

CPU Profiling

- Objectives:
 - Profile CPU usage by stack sampling
 - Generate CPU flame graphs
- Use perf/perf_events
- Other tools:
 - Gprof
 - Vtune

Perf

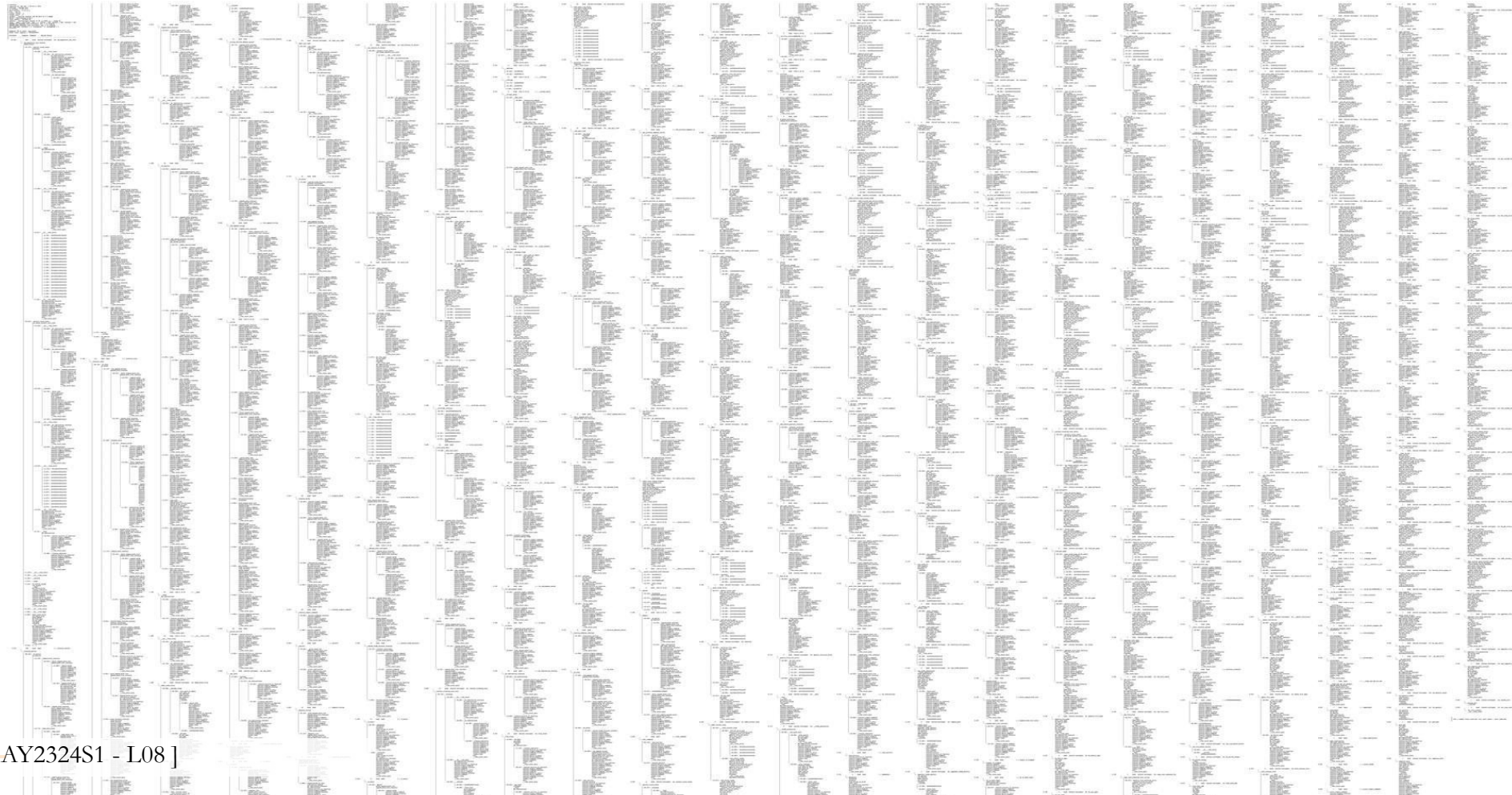
- Modern architectures expose performance counters
 - ❑ Cache misses, branch mispredictions, IPC, etc
- Perf tool provides easy access to these counters
 - ❑ perf list – list counters available on the system
 - ❑ perf stat – count the total events
 - ❑ perf record – profile using one event
 - ❑ perf report – Browse results of perf record

perf_events

- **Multi-tool** with many capabilities
 - ❑ CPU profiling
 - ❑ Cache profiling
 - ❑ Static & dynamic tracing
- In Linux source code: tools/perf
 - ❑ Usually pkg added by linux-tools-common, etc.

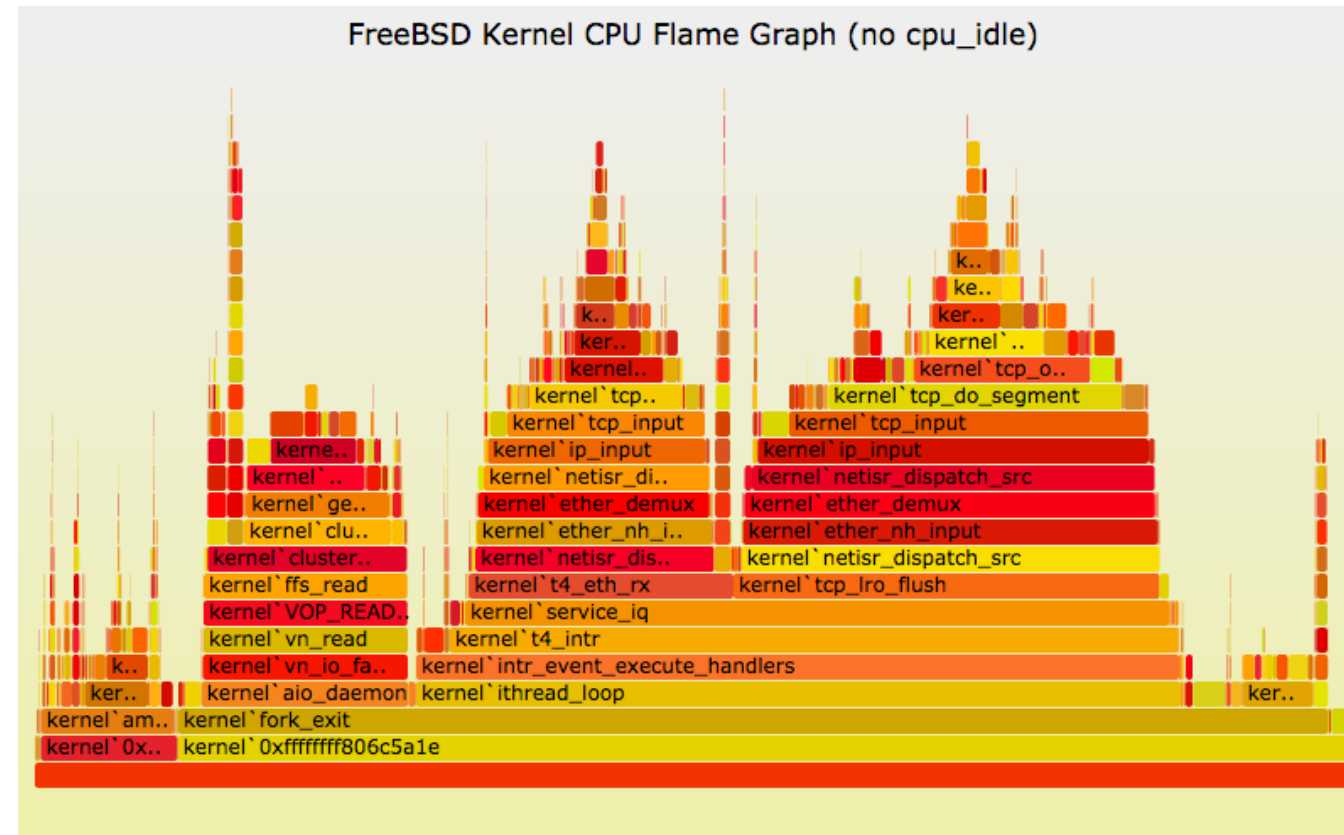
perf_events: Full “Report” Output

- `perf record -F 99 -p 13204 -g -- sleep 30`
- `perf report -n --stdio`



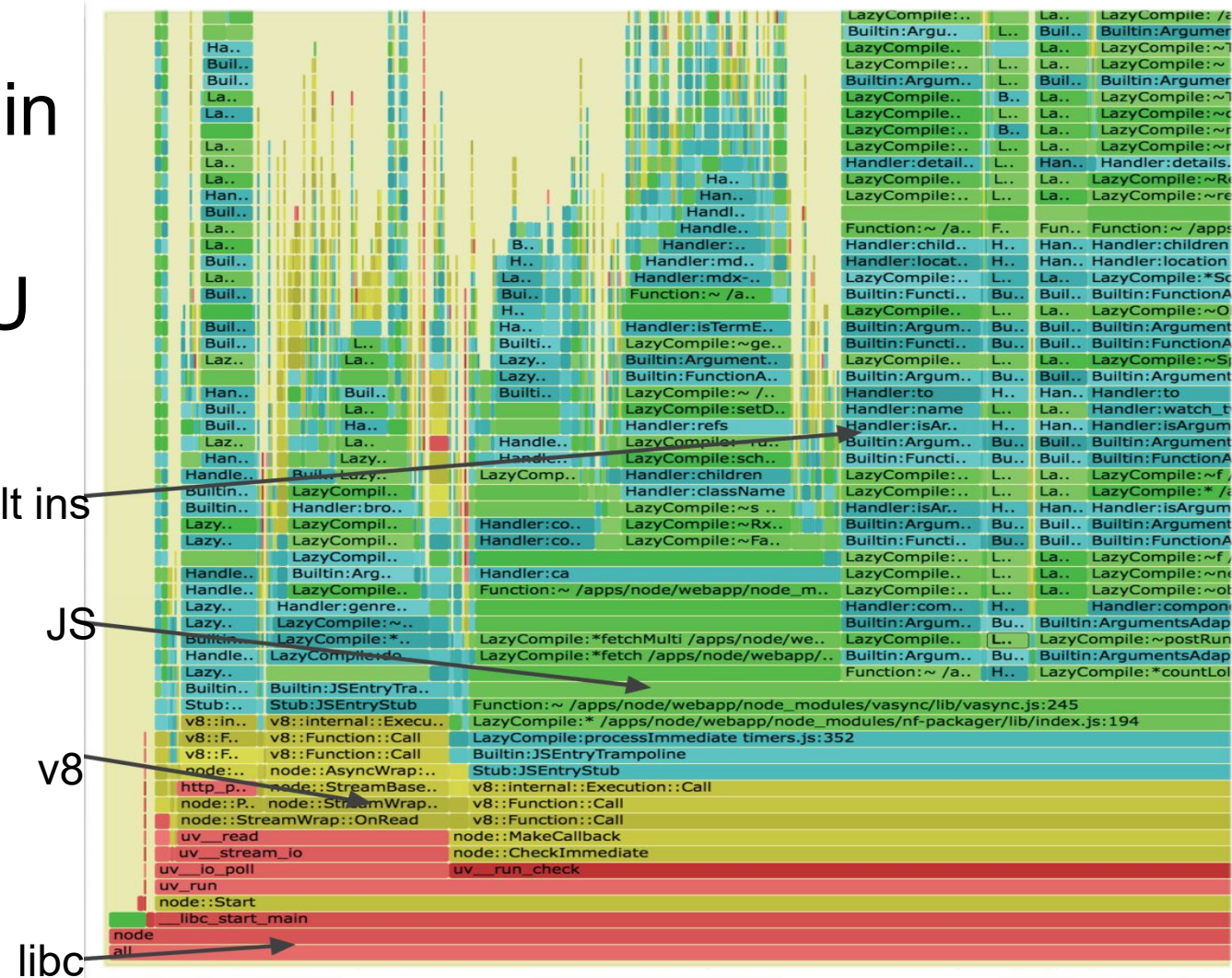
CPU Profile Method

- Understand all software in CPU profile > 1%
- Discovers a wide range of issues by CPU usage
 - ❑ Directly: CPU consumers
 - ❑ Indirectly: initialization of I/O, locks, times, ...
- Narrows down software to study
- Flame graphs →

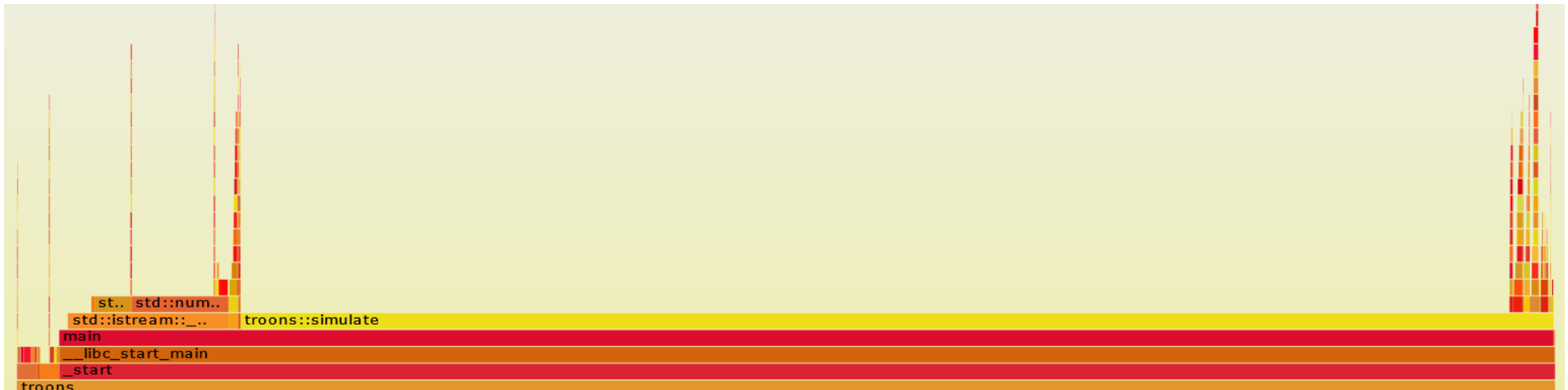
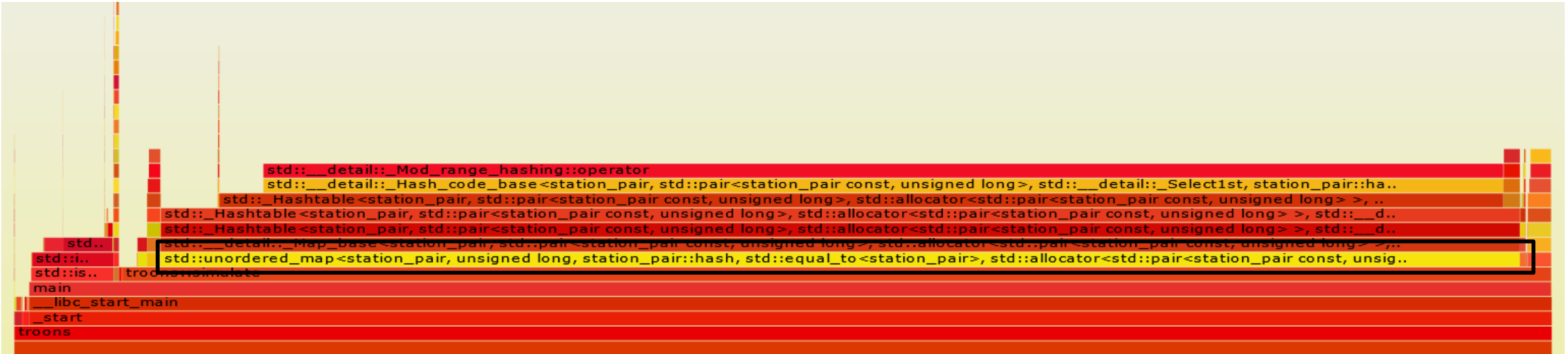


Profiling with Flame Graphs

- Each box presents a function in the stack (stack frame)
- x-axis: percent of time on CPU
- y-axis: stack depth
- colors: random, or can be a dimension



Assignment 1 (last year) Sequential Implementation



Understanding System's Performance

- Perspectives
 - Resources
 - Workload
- Methodologies
- Observability Tools
 - Counters and Profiling
 - Debugging Tools
- Benchmarking

Debugging Tools

- Help in identifying bugs

- Valgrind

- Heavy-weight binary instrumentation: >4x overhead
 - Designed to shadow all program values: registers and memory
 - Shadowing requires serializing threads
 - Usually used for memcheck

- Sanitizers

- Compilation-based approach

Valgrind memcheck

- Validates memory operations in a program
 - ❑ Each allocation is freed only once
 - ❑ Each access is to a currently allocated space
 - ❑ All reads are to locations already written
 - ❑ 10 – 20x overhead

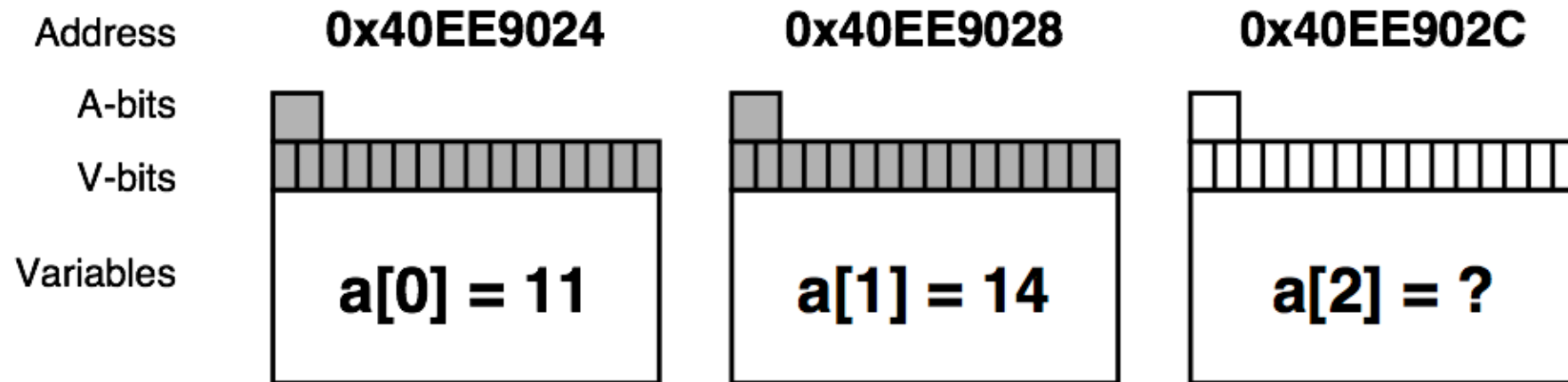
```
valgrind --tool=memcheck <prog ...>
```

```
...
==29991== HEAP SUMMARY:
==29991==      in use at exit: 2,694,466,576 bytes in 2,596 blocks
==29991==    total heap usage: 16,106 allocs, 13,510 frees, 3,001,172,305 bytes allocated
==29991==
==29991== LEAK SUMMARY:
==29991==    definitely lost: 112 bytes in 1 blocks
==29991==    indirectly lost: 0 bytes in 0 blocks
==29991==    possibly lost: 7,340,200 bytes in 7 blocks
==29991==    still reachable: 2,687,126,264 bytes in 2,588 blocks
==29991==           suppressed: 0 bytes in 0 blocks
```

Valgrind – How to?

■ Shadow memory

- ❑ Used to track and store information on the memory that is used by a program during its execution.
- ❑ Used to detect and report incorrect accesses of memory



A-bit: corresponding byte accessible

V-bit: corresponding bit initialized

Sanitizers

- Compilation-based approach to detect issues
 - ❑ GCC and LLVM support
 - ❑ ~2x overhead
 - ❑ Add “-fsanitize=address”
- Examples:
 - ❑ ThreadSanitizer (e.g., data races)
 - ❑ MemorySanitizer (e.g., uninitialized reads)
 - ❑ UndefinedBehaviorSanitizer (e.g., Integer Overflow, Null Pointer)
 - ❑ LeakSanitizer (e.g., memory leaks)

-fsanitize=address Example

```
$ cat test.c
int main(int argc, char **argv) {
    int a[2] = {11, 14};
    return a[2];
}
$ gcc -fsanitize=address -o test test.c
$ ./test
=====
==14504== ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff578920e8 at pc
0x400844 bp 0x7fff578920b0 sp 0x7fff578920a8
READ of size 4 at 0x7fff578920e8 thread T0
    #0 0x400843 (/home/ubuntu/test+0x400843)
    #1 0x7fbfceb57ec4 (/lib/x86_64-linux-gnu/libc-2.19.so+0x21ec4)
    #2 0x400688 (/home/ubuntu/test+0x400688)
Address 0x7fff578920e8 is located at offset 40 in frame <main> of T0's stack:
  This frame has 1 object(s):
    [32, 40) 'a'
Shadow bytes around the buggy address:
  0x10006af0a400: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x10006af0a410: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00[f4]f4 f4
  0x10006af0a420: f3 f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
```

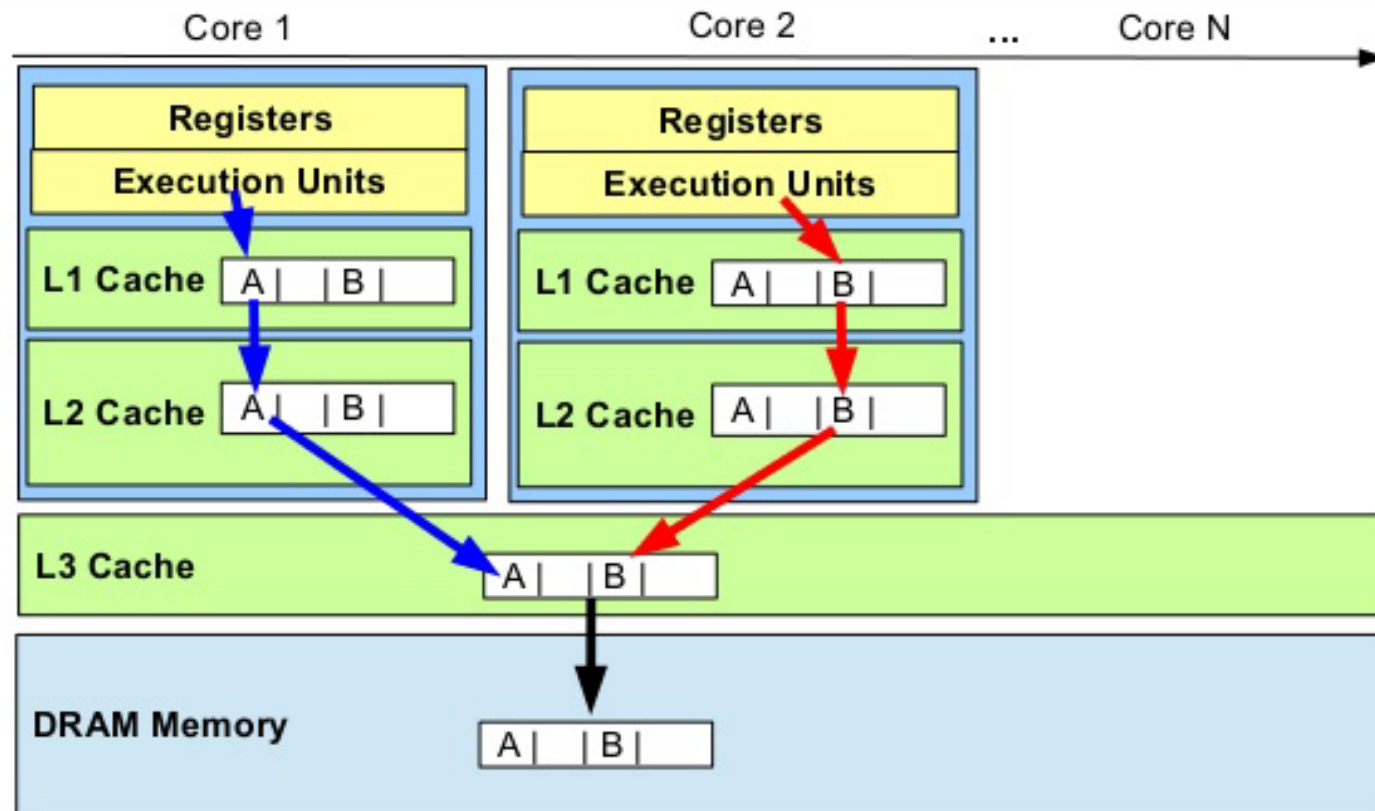
-fsanitize=thread Example

```
#include <pthread.h>
#include <stdio.h>
int global;
void *Thread1(void *x) {
    global++;
    return NULL;
}
void *Thread2(void *x) {
    global--;

    return NULL;
}
int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    pthread_create(&t[1], NULL, Thread2, NULL);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
}
```

```
=====
WARNING: ThreadSanitizer: data race (pid=17631)
    Read of size 4 at 0x562a4b132014 by thread T2:
        #0 Thread2 <null> (thread_san+0xa42)
        #1 <null> <null> (libtsan.so.0+0x296ad)
    Previous write of size 4 at 0x562a4b132014 by
    thread T1:
        #0 Thread1 <null> (thread_san+0xa03)
        #1 <null> <null> (libtsan.so.0+0x296ad)
    Location is global 'global' of size 4 at
    0x562a4b132014 (thread_san+0x000000202014)
    Thread T2 (tid=17635, running) created by main
    thread at:
        #0 pthread_create <null> (libtsan.so.0+0x2bcee)
        #1 main <null> (thread_san+0xad3)
    Thread T1 (tid=17634, finished) created by main
    thread at:
        #0 pthread_create <null> (libtsan.so.0+0x2bcee)
        #1 main <null> (thread_san+0xab2)
SUMMARY: ThreadSanitizer: data race
(/home/ccris/GitRepos/CS3210-AY2021-
S1/Lectures/perf_inst/thread_san+0xa42) in Thread2
=====
ThreadSanitizer: reported 1 warnings
```

Problem: False Sharing



```
struct foo {  
    int x;  
    int y;  
};  
  
static struct foo f;
```

```
int sum_a(void)  
{  
    int s = 0;  
    for (int i = 0; i < 1000000; ++i)  
        s += f.x;  
    return s;  
}
```

```
void inc_b(void)  
{  
    for (int i = 0; i < 1000000; ++i)  
        ++f.y;  
}
```


Identifying False Sharing using Perf

- Use `perf c2c`
- At a high level it shows:
 - ❑ The cache lines where false sharing was detected.
 - ❑ The readers and writers to those cache lines, and the offsets where those accesses occurred.
 - ❑ The pid, tid, instruction addr, function name, binary object name for those readers and writers.
 - ❑ The source file and line number for each reader and writer.
 - ❑ The average load latency for the loads to those cachelines.

Understanding System's Performance

- Perspectives
- Methodologies
- Instrumentation Tools
 - Observability Tools and Profiling
 - Debugging Tools
- **Benchmarking**

Benchmarking Tools

- ~100% of benchmarks are wrong
- Results are usually misleading
 - ❑ you benchmark A, but actually measure B, and conclude you measured C
- Common mistakes:
 - ❑ Testing the wrong target: eg, FS cache instead of disk
 - ❑ Choosing the wrong target: eg, disk instead of FS cache ... doesn't resemble real world usage
 - ❑ Invalid results: bugs
- The energy needed to refute benchmarks is multiple orders of magnitude bigger than to run them

Benchmark Program: **Overview**

- It is **not easy** to evaluate and compare the performance of computer systems
 - Our discussion focused only on processor and memory performance
 - ➔ Much more complicated if we bring in other aspects of a system
- A few well-known benchmarks

Benchmarks: **Industry Standards**

■ **SPEC benchmark suites:**

- ❑ **SPECint, SPECfp**: For processor + memory + compiler
- ❑ **SPECjvm2008**: For Java performance
- ❑ Many others

■ **EEMBC benchmark suites**

- ❑ **ANDBench**: For Android performance
- ❑ **DPIBench**: For system and network performance
- ❑ etc

■ **Numerical Aerodynamic Simulation (NAS):**

- ❑ From NASA
- ❑ Massively parallel benchmark: For computer cluster

Benchmarks: Simple Benchmark

These benchmarks can be found easily on the web

- **Linpack:**

- ❑ Linear Algebra Solver
- ❑ Used in the SuperComputer ranking

- **Dhrystone / Whetstone:**

- ❑ Small program to test integer / floating performance

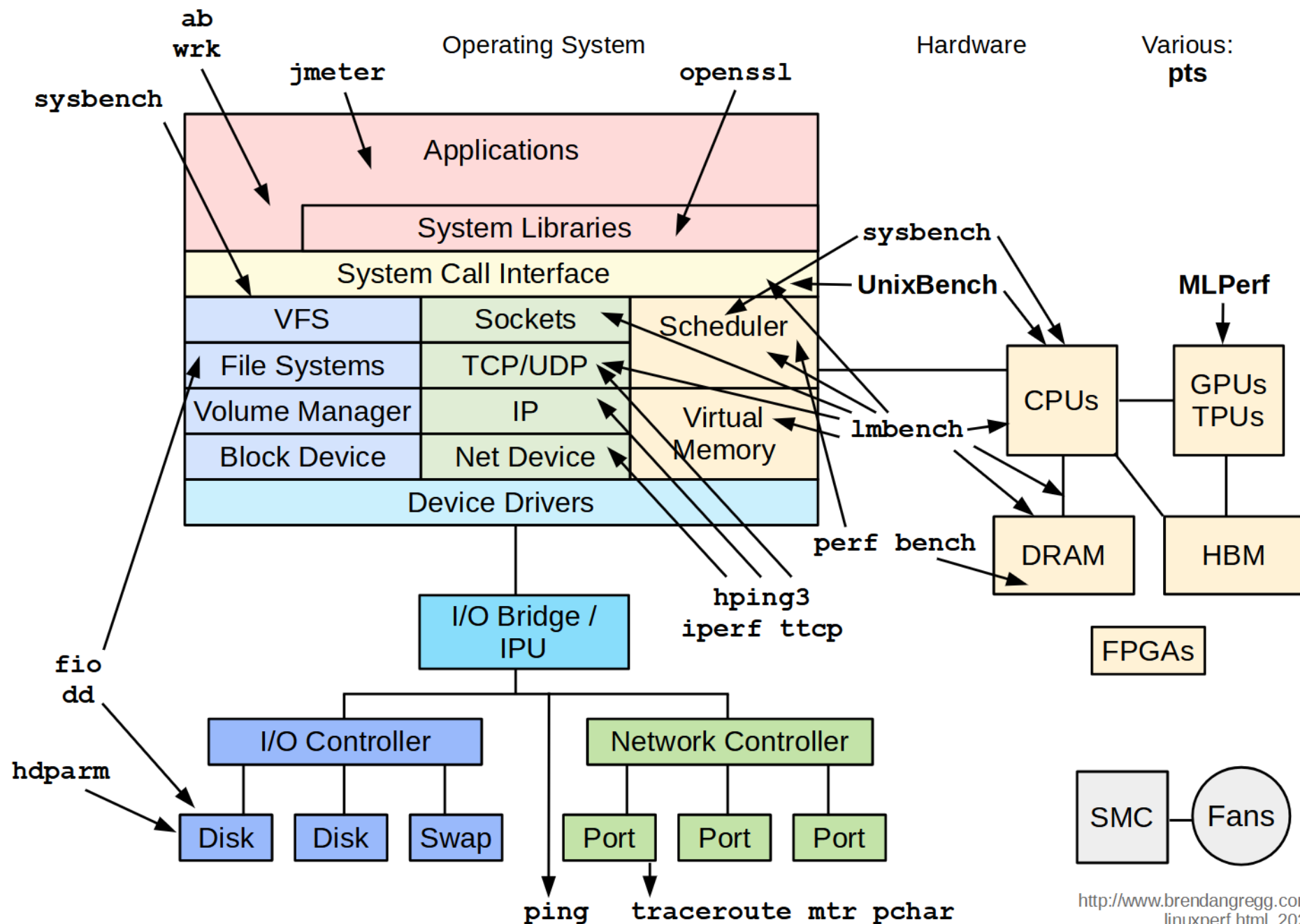
- **Tak Function:**

- ❑ To test recursion optimization

Active Benchmarking Method (Synthetic Performance Testing)

- Run the benchmark for hours
- While running, analyze and confirm the performance limiter using *observability tools*
 - Disk benchmark: run iostat, ...
 - CPU benchmark: run pidstat, perf, flame graphs, ...
- Answer the question: why isn't the result 10x better?

Linux Performance Benchmark Tools



Summary

- Performance instrumentation needs an understanding of resource utilization
 - Methodologies
 - Tools to observe performance
 - Benchmarks

References

- Reading: Brendan Gregg, Netflix - <http://www.brendangregg.com/linuxperf.html>
- Further reading:
 - ❑ <http://queue.acm.org/detail.cfm?id=1117403>