P56

Lecture five :

Friday 15:00-17:00 SE02
Andrew.Moore@cl.cam.ac.uk

Existing tools

Tools

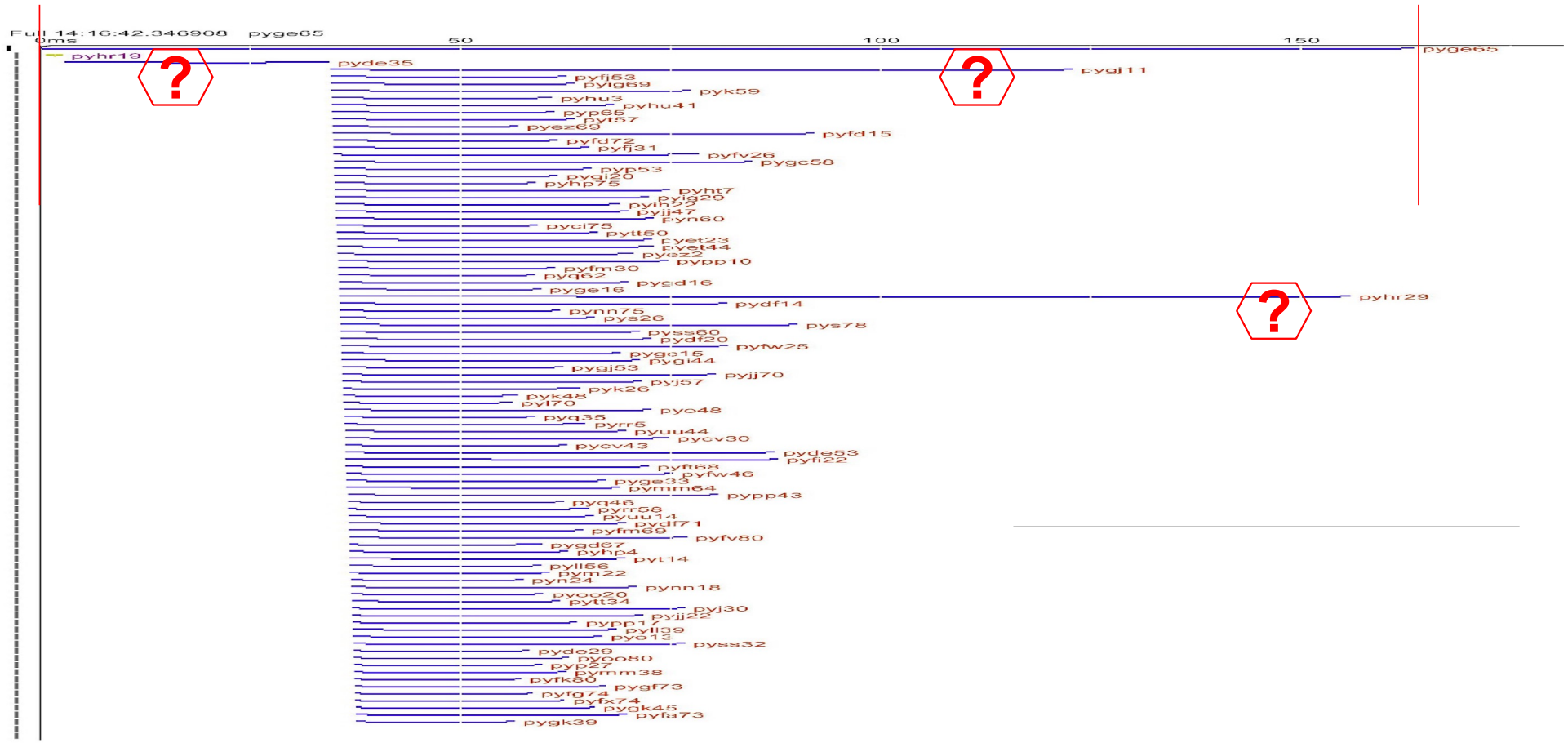
logs
dashboards
top
strace
blktrace
gprof
oprofile
valgrind
perf
vtune

Logs

Logs are the basis for tracking where the time went within many layers of software, tracking offered load abuses, and tracking sub-service health.

```
start_time   end_time   rpcid   parent   from   to   req_len   resp_len   method   ...
start_time   end_time   rpcid   parent   from   to   req_len   resp_len   method   ...
start_time   end_time   rpcid   parent   from   to   req_len   resp_len   method   ...
 ...
```

Constructed from logs



Dashboards

Dashboards display for a given service, sub-service, server, etc. current behavior

```
              Last
Req/second  10sec 1min  10min
Total         xxx           xxx           xxx
  read        xxx           xxx           xxx
  write             xxx           xxx           xxx
  open            xxx             xxx           xxx
  ...
Timeouts    xxx           xxx           xxx
Errors            xxx           xxx           xxx
 ...
```

top

top shows a server's current top processes every few seconds, memory use, user/system/idle, etc.

```
top - 14:35:01 up 5 days, 22:30,  4 users,  load average: 0.07, 0.06, 0.01
Tasks: 170 total,   1 running, 169 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.8 us,  0.0 sy,  0.0 ni, 98.8 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  3920956 total,   680524 free,  1115880 used,  2124552 buff/cache
KiB Swap:  4067324 total,  3754784 free,   312540 used.  2481128 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3045 sites     20   0  658940  33776  21724 S   1.0  0.9   0:29.15 gedit
18280 sites     20   0 1023180 129688  60960 S   0.7  3.3   0:22.57 chrome
  211 root      20   0   32088   3780   3584 D   0.3  0.1   0:01.38 systemd-journal
 2943 sites     20   0  101484   5036   1416 S   0.3  0.1   3:16.64 sshd
18369 sites     20   0   41800   3376   2788 R   0.3  0.1   0:00.30 top
    1 root      20   0  119732   4336   2848 S   0.0  0.1   0:03.43 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.02 kthreadd
```

strace

strace shows all system calls -- very slow but sometimes informative.

$ strace ls -l

```
execve("/bin/ls", ["ls", "-l"], [/* 73 vars */]) = 0
brk(NULL)                                       = 0x1e8b000
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd81b52b000
access("/etc/ld.so.preload", R_OK)              = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=100934, ...}) = 0
mmap(NULL, 100934, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd81b512000
close(3)                                        = 0
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260Z\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
```

blktrace

blktrace tracks disk activity. It unfortunately needs sudo in order to run.

$ blktrace

$ blkparse sda2

```
Input file sda2.blktrace.0 added
Input file sda2.blktrace.1 added
Input file sda2.blktrace.3 added
  8,0    0        1     0.000000000 22106  A   W 55314672 + 8 <- (8,2) 54264048
  8,2    0        2     0.000000853 22106  Q   W 55314672 + 8 [kworker/u8:2]
  8,2    0        3     0.000004372 22106  G   W 55314672 + 8 [kworker/u8:2]
  8,2    0        4     0.000004776 22106  P   N [kworker/u8:2]
  8,2    0        5     0.000007113 22106  I   W 55314672 + 8 [kworker/u8:2]
  8,2    0        6     0.000007640 22106  U   N [kworker/u8:2] 1
  8,2    0        7     0.000008276 22106  D   W 55314672 + 8 [kworker/u8:2]
  8,0    1        1     0.000022306   178  A  WS 240630760 + 8 <- (8,2) 239580136
  8,2    1        2     0.000022763   178  Q  WS 240630760 + 8 [jbd2/sda2-8]
  8,2    1        3     0.000024418   178  G  WS 240630760 + 8 [jbd2/sda2-8]
  8,2    1        4     0.000024608   178  P   N [jbd2/sda2-8].
```

gprof

gprog generates profiles of single-program execution, for C/C++ programs compiled with -pg

$ gprof stable_task gmon.out

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
81.21	0.30	0.30				std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>,
13.54	0.35	0.05				std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>,
5.41	0.37	0.02				frame_dummy
0.00	0.37	0.00	4096	0.00	0.00	(anonymous namespace)::CharToBase40(char const*)
0.00	0.37	0.00	1	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>,

...

oprofile

oprofile profiles an entire Linux system, using unmodified binaries.

$ opreport --exclude-dependent

```
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events count 50000
   450385 75.6634 cc1plus
    60213 10.1156 lyx
    29313  4.9245 XFree86
    11633  1.9543 as
    10204  1.7142 oprofiled
     7289  1.2245 vmlinux
     7066  1.1871 bash
     6417  1.0780 oprofile
     6397  1.0747 vim
     3027  0.5085 wineserver
     1165  0.1957 kdeinit
      832  0.1398 wine
```

valgrind

The valgrind tool is mostly used to interpret programs slowly and check carefully for memory errors and leaks. It also can profile programs.

```
void f(void) {
  int* x = malloc(10 * sizeof(int));
   x[10] = 0;         // problem 1: heap block overrun
}                     // problem 2: memory leak -- x not freed
int main(void) { f();  return 0; }
```

Most error messages look like the following, which describes problem 1, the heap block overrun:

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

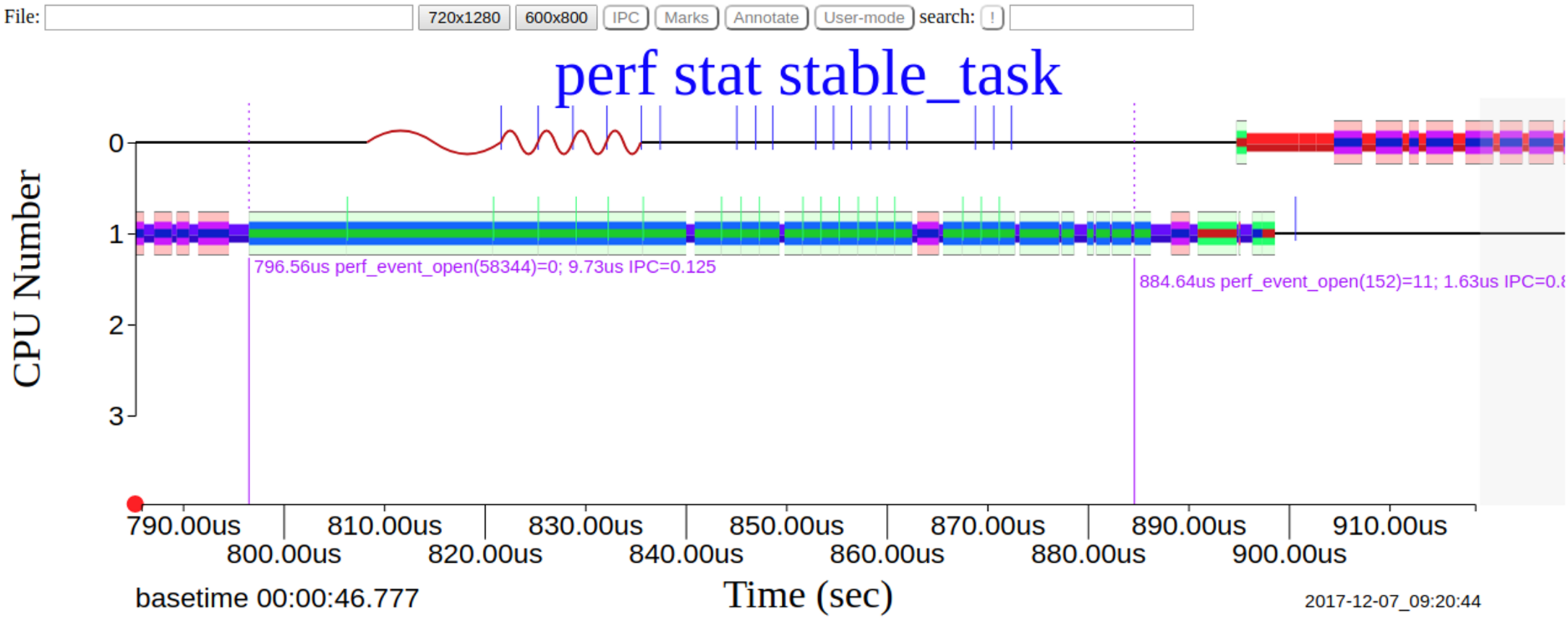
perf

perf is a package that uses the x86 performance counters to give statistics about a running program

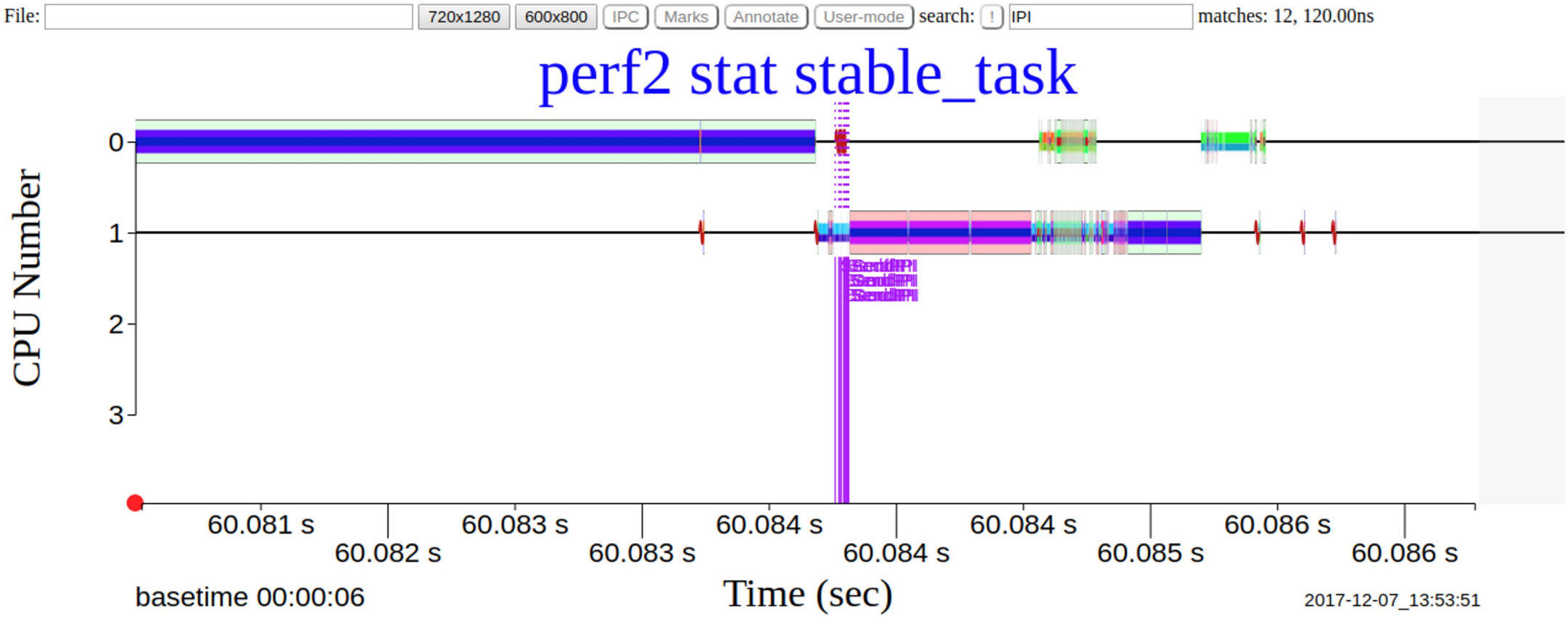
$ ./perf stat ./stable_task stable_data

```
Total on 1024 blocks 11.514054 sec
 Performance counter stats for './stable_task stable_data':
      11986.736265      task-clock (msec)         #    1.000 CPUs utilized
               40      context-switches          #    0.003 K/sec
                0      cpu-migrations            #    0.000 K/sec
            74803      page-faults               #    0.006 M/sec
      33560077965      cycles                    #    2.800 GHz
   <not supported>     stalled-cycles-frontend
   <not supported>     stalled-cycles-backend
      12872513081      instructions              #    0.38  insns per cycle
       3695416330      branches                  #  308.292 M/sec
        188304899      branch-misses             #    5.10% of all branches
11.987501391 seconds time elapsed
```

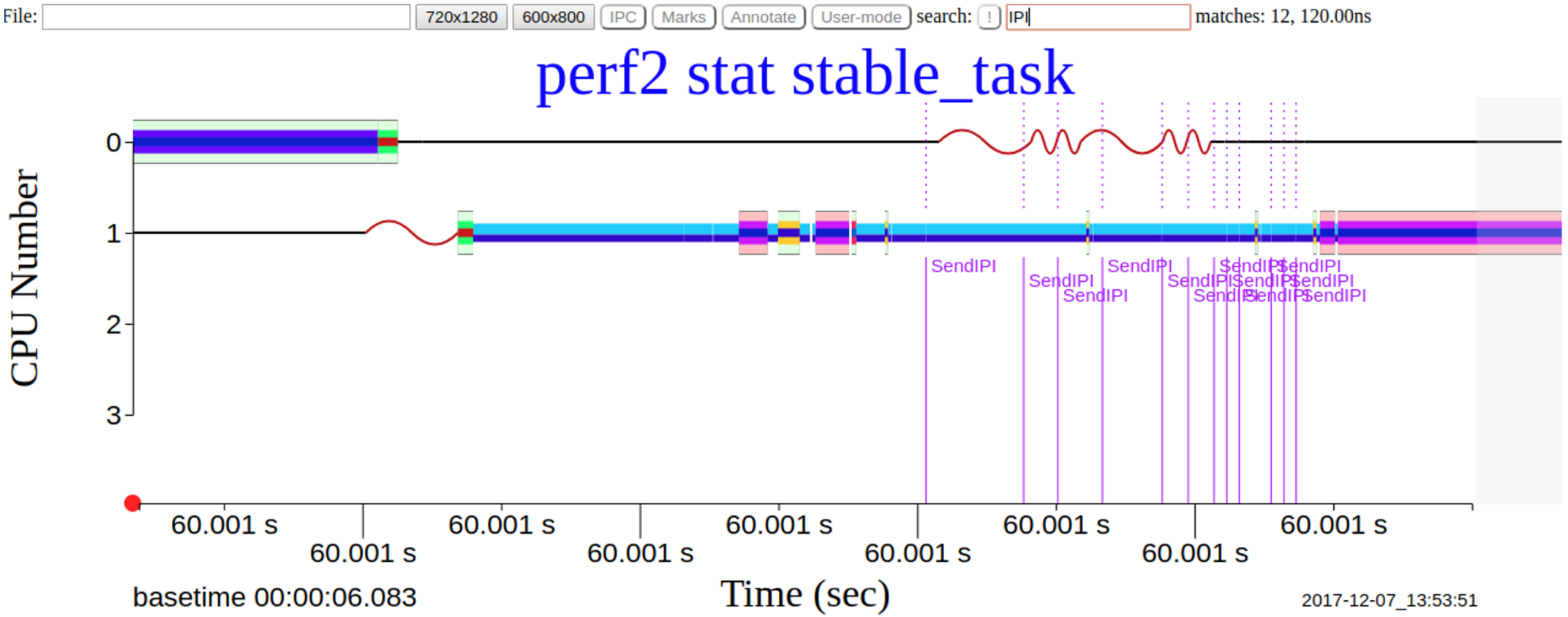
perf getting started -- 100us perf_event_open calls



perf ending -- IPIs



perf ending -- IPIs



vtune

vtune is somewhat like perf, but Intel-specific. It costs $800+, but is used commercially to tune some heavy-use software.

Counts

Counters

Some Ethernet counters from ifconfig:

```
RX packets 241247009  bytes 93774363043 (93.7 GB)
RX errors 0  dropped 14645  overruns 0  frame 0

TX packets 56231194  bytes 66349781183 (66.3 GB)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Counters are good cheap observation tools, but they do not tell you *why*.

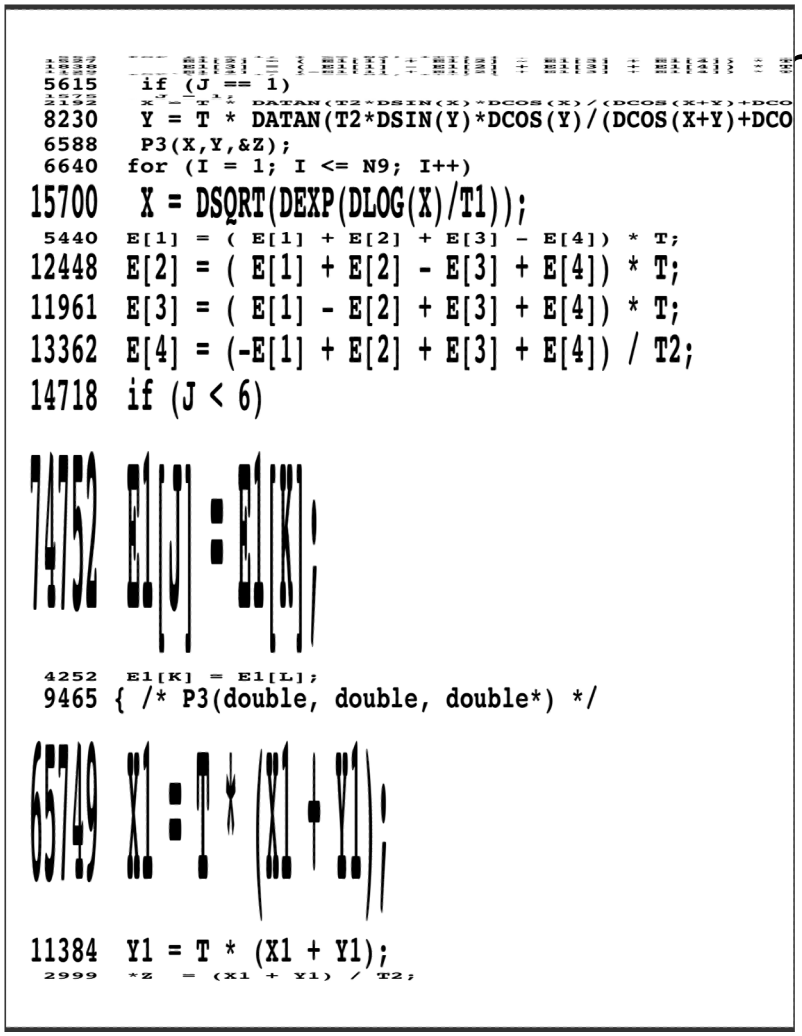
Profiles

Profiling a bad benchmark (circa 1972)

Whetstone -- synthetic code to match the frequency of 124 instructions while running some 1960s physics programs on the KDF9 computer (UK, circa 1964).

- Nominally a test of floating-point arithmetic performance
- It actually measured memory accesses
- The main-program-only profile is seriously misleading

Whetstone Profile



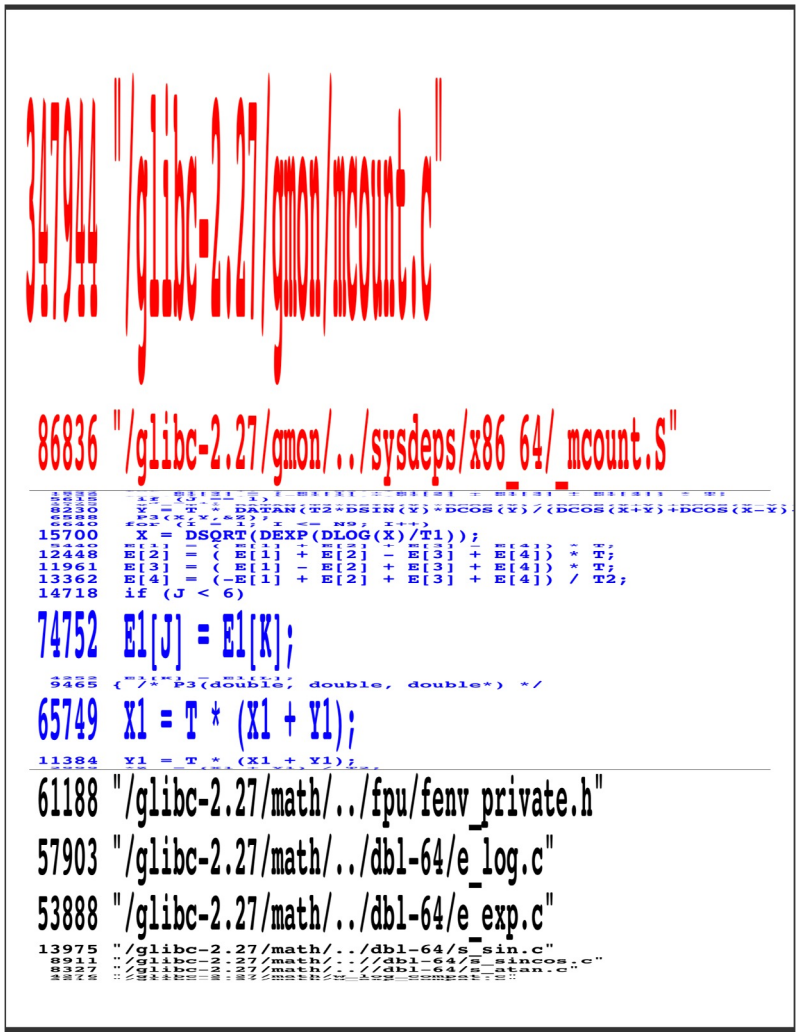
Full whetstone Profile

Main program is only 1/3 of runtime

The profiling mechanism **mcount** is about 40%

The library routines for **fenv, log, exp, sin, etc.** are about 25%

We don't know anything about operating system time



Profiles

Profiles are a good cheap tool for understanding *average* behavior of user-mode programs.

Blind to kernel time

Blind to the differences between slow work and normal work.

Useless for understanding root causes.

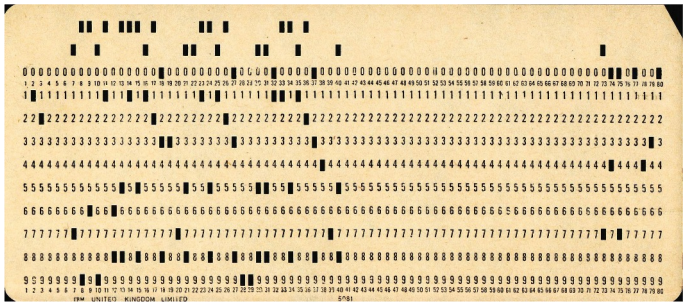
Traces

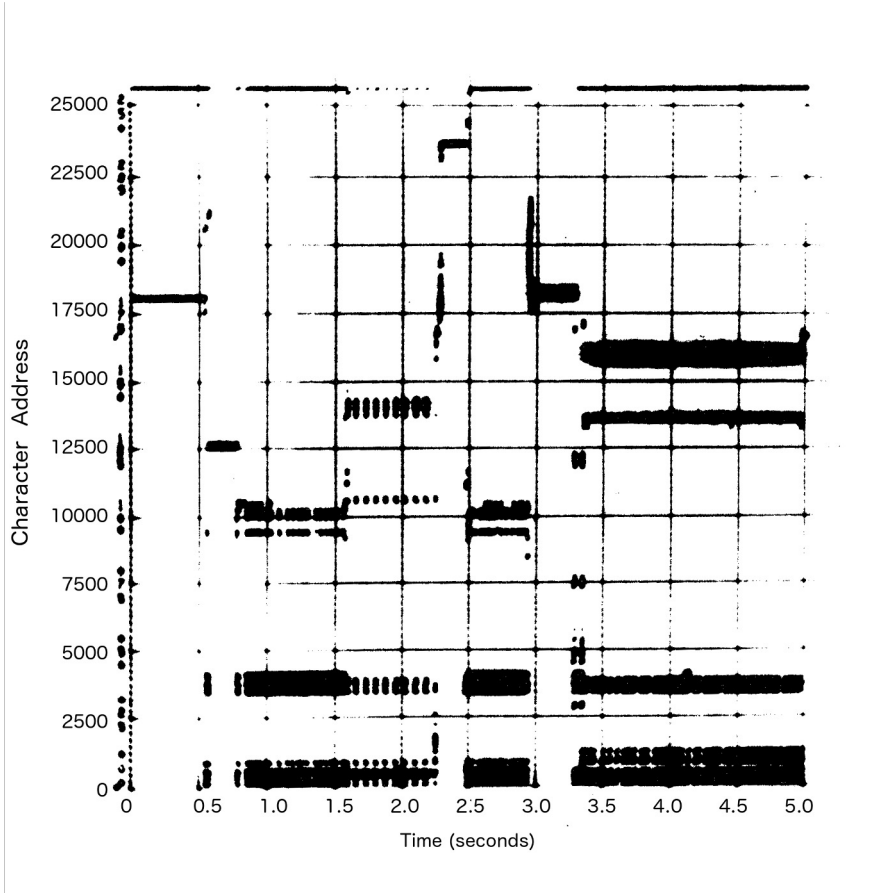
IBM 7010 instruction trace

Five seconds of sort-program startup

Loads program from tape/disk

Reads control parameters from cards

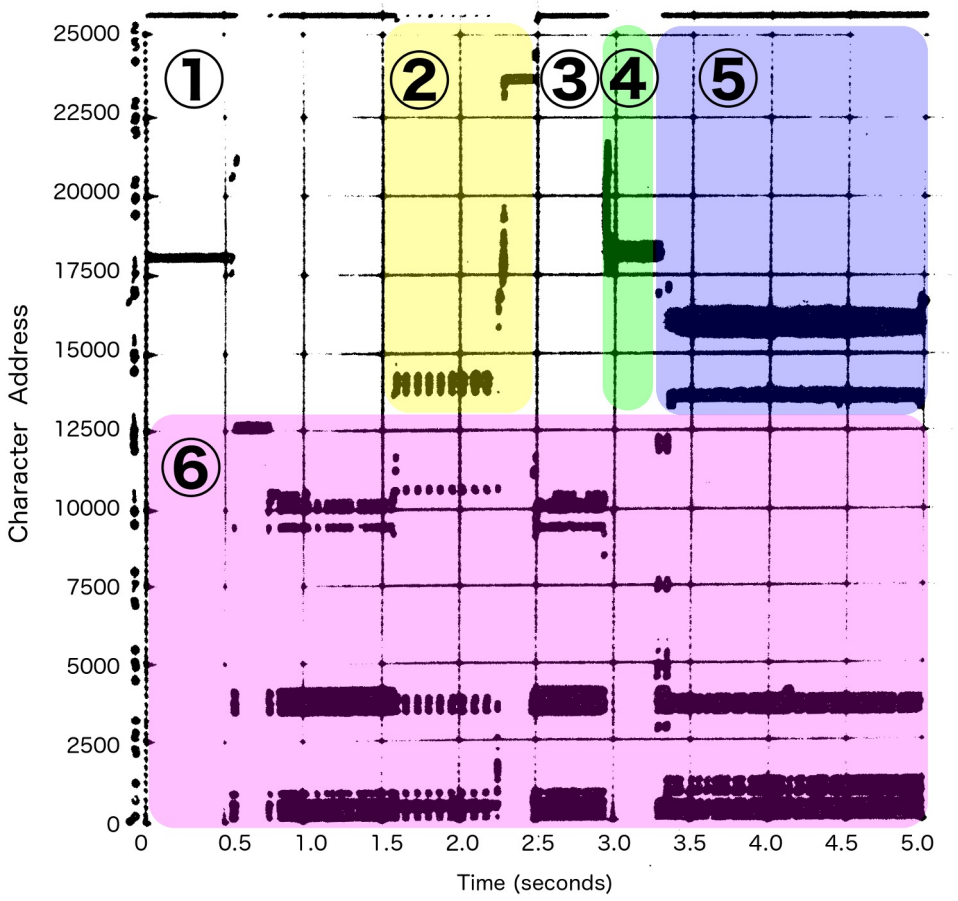




Instruction trace

1 Load the monitor (op.syst.)
  Load sort program first overlay
2 Sort reads 9 control cards
3 Load sort program second overlay
4 Initialize real sorting
5 Read first few records

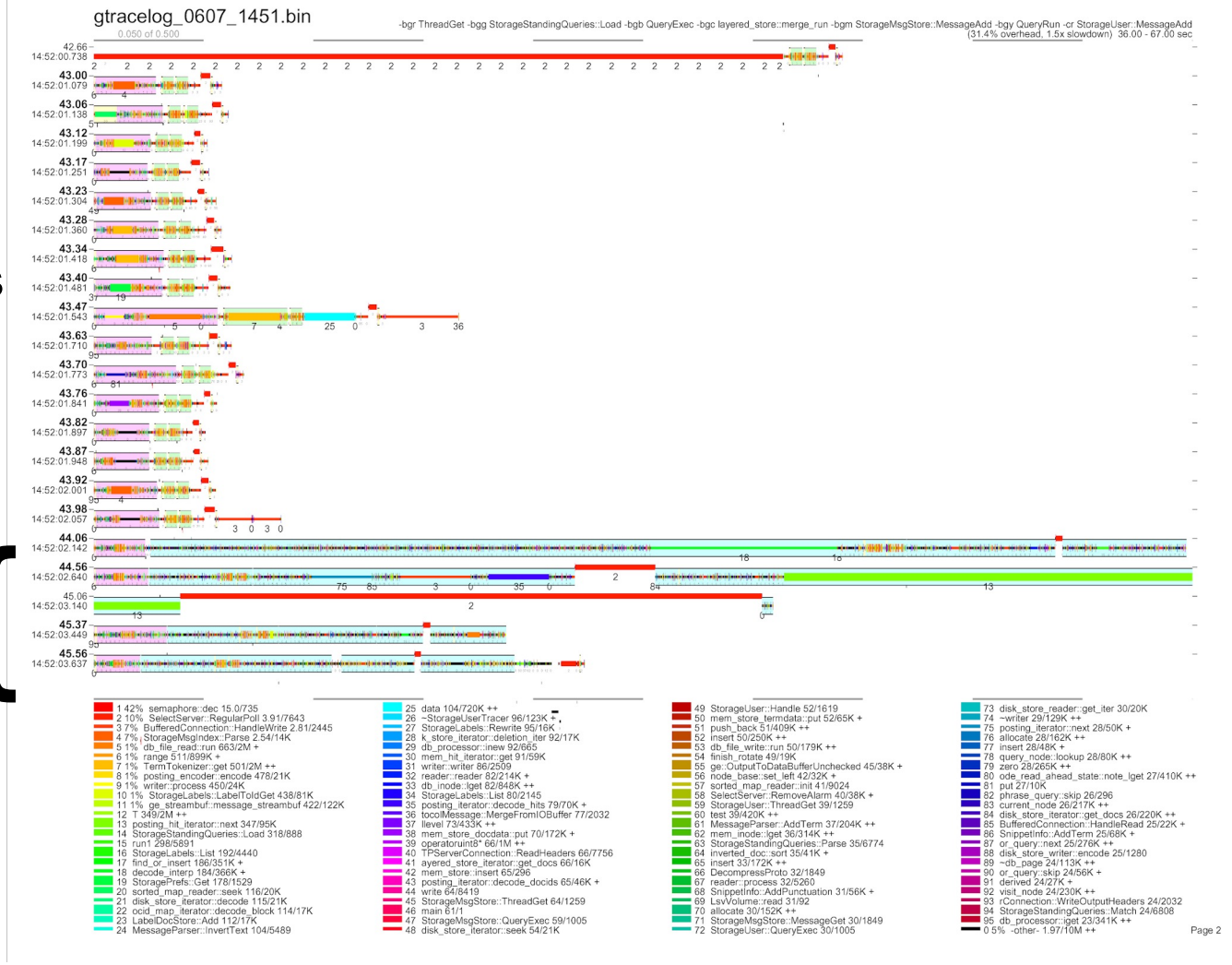
6 The monitor, 12,600 characters



Gmail call/ret trace

16 Normal message uploads

Three slow messages



Gmail trace

We can see the slow transactions and therefore see what is different about them compared to normal ones.

That *immediately* reveals what to fix - - what the unexpected dynamic is.

Tracing summary

Traces are records of time-sequenced events, such as all the accesses on a disk drive or all the system calls in a program. Usually, the individual events are timestamped to the microsecond or better.

Tracing is the only tool that can capture unexpected interactions or interference between programs --

You don't know *when* or *where* to look, so you have to look for a long-ish time period at everything that is happening.

The Four Fundamental Resources, plus one

CPU

Memory

Disk/SSD

Network

Software locks

CPU

Memory

Disk/SSD

Network

Software locks

If a transaction we care about is slow because of <**CONSTRAINED RESOURCE>** interference, what can we observe to identify that as the source of interference?

Performance mysteries

It is often easier to do experiments making software **slower** instead of making it **faster**.

Technique #1: Run suspect code **twice** instead of once. If things get noticeably slower, then the suspect code matters; if no change then look elsewhere.

Technique #2: Run interference hogs that deliberately consume large amounts of one of the five shared resources, alternating with idle. See if performance problems correlate with the non-idle times. Look especially at the program dynamics at the start/stop *transitions*.

Enter the Hog

Antagonistic workloads

Performance mystery hogs

- Consider the five self-contained programs below. Each should run for several minutes (or until kill -9 is sent to it), creating interference in a tight loop (IATL) for approximately 1/4 of a second then doing nothing for 1/4 of a second. Repeat.

- It should be possible to run several copies of each program at once (perhaps using different command-line parameters) and to run different programs at once.

Performance mystery hogs

- **cpu_hog_xxxx** -- use up one complete CPU core IATL without touching memory
- **l2_hog_xxxx** -- access the L2 cache IATL without touching L3 and without using more CPU time than necessary
- **dram_hog_xxxx** -- access main memory IATL, without using much CPU time **disk_hog_xxxx** -- read from disk in IATL, avoiding reading from in-memory disk
- cache (write ~40MB at the beginning and read from there)
- **network_hog_xxxx** -- transmit 1MB messages repeatedly IATL to a second server

(IATL = In a Tight Loop)

Performance mystery hogs

- Consider the five self-contained programs below. Each should run for several minutes (or until kill -9 is sent to it), creating interference in a tight loop (IATL) for approximately 1/4 of a second then doing nothing for 1/4 of a second. Repeat.

- It should be possible to run several copies of each program at once (perhaps using different command-line parameters) and to run different programs at once.

Performance mystery hogs

Lab 5 is all about CPU

Other hogs are available.

Past examples of code explored

- Generate training data for a game AI (written in Go)
- Parallel n-body with OpenMP
- A simple web server
- Examining MMAP vs Non-MMAP reading of large out-of-core data-structures
- HM: some code I found on Github. (bash history manager)
- KiCad Library Utils
- Using a KUTrace Marking in Golang
- server4-client4
- git status mystery
- Improving read performance of randomly located, small contiguous blocks, in a large dataset
- Multi-threaded Read and Write
- Kutrace on 32 arm for a real time problem
- Study server_disk
- Tracing gRPC
- Dynamic MPI Mandelbrot Program
- Tracing Ligra
- Spinlock Analysis client servers from Lab4
- Map-reduce implementation of MPI
- ed25519
- Raft consensus algorithm stuff
- KUTracing a ray tracer
- Measuring a minibrowser