Advanced Systems Lab

G. Alonso, D. Kossmann

Systems Group

http://www.systems.ethz.ch/

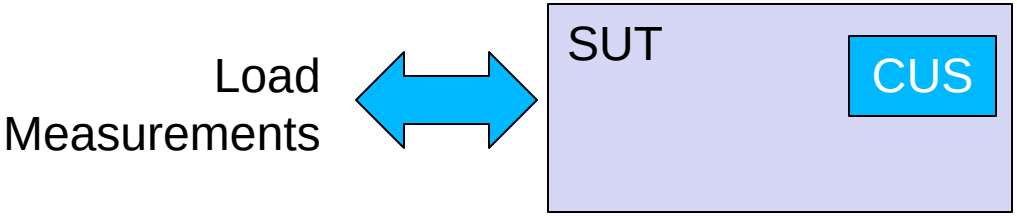
This week: deeper into workloads

1. Workload selection
   - Where do these benchmarks come from?
   - At what level do you measure?
2. Workload characterization
   - How do you decide what load to put in?
3. Monitoring and Instrumentation
   - How do you get the information out?

Workload selection

Services

- Don’t confuse:
  - the System Under Test (SUT)
  - the Component Under Study (CUS)
- We apply load to the SUT
- We measure performance of SUT
- We want to understand the CUS!
- Also: beware of external components…



Example: CPU performance

Systems@ETH zürich

- SPECint often used to compare CPUs
  - CUS: the CPU itself
  - BUT: SUT is the whole computer!

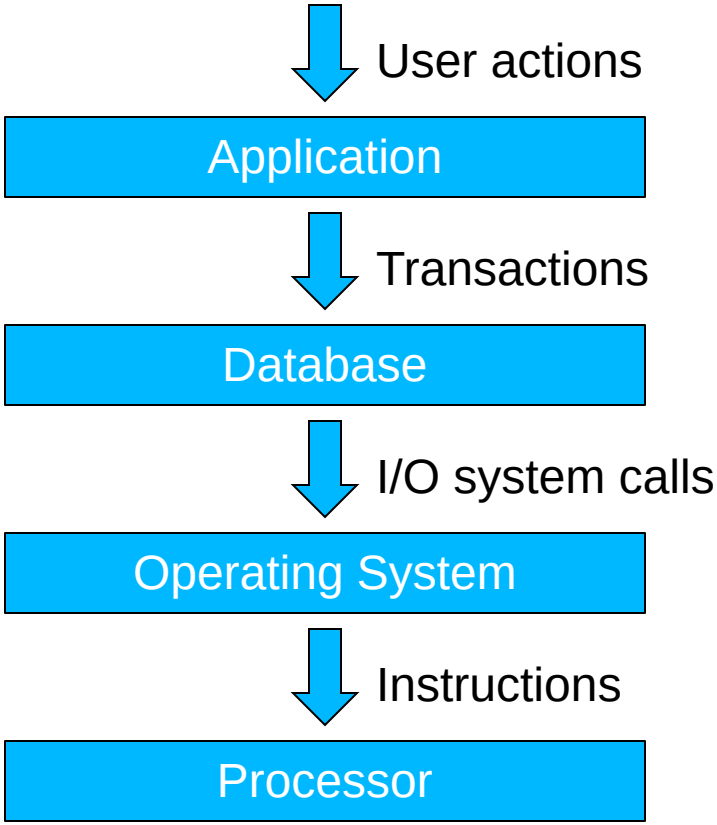
- For your Lab exercises:
  - What's the CUS?
  - What's the SUT?
  - Are there any external components?

Systems provide Services

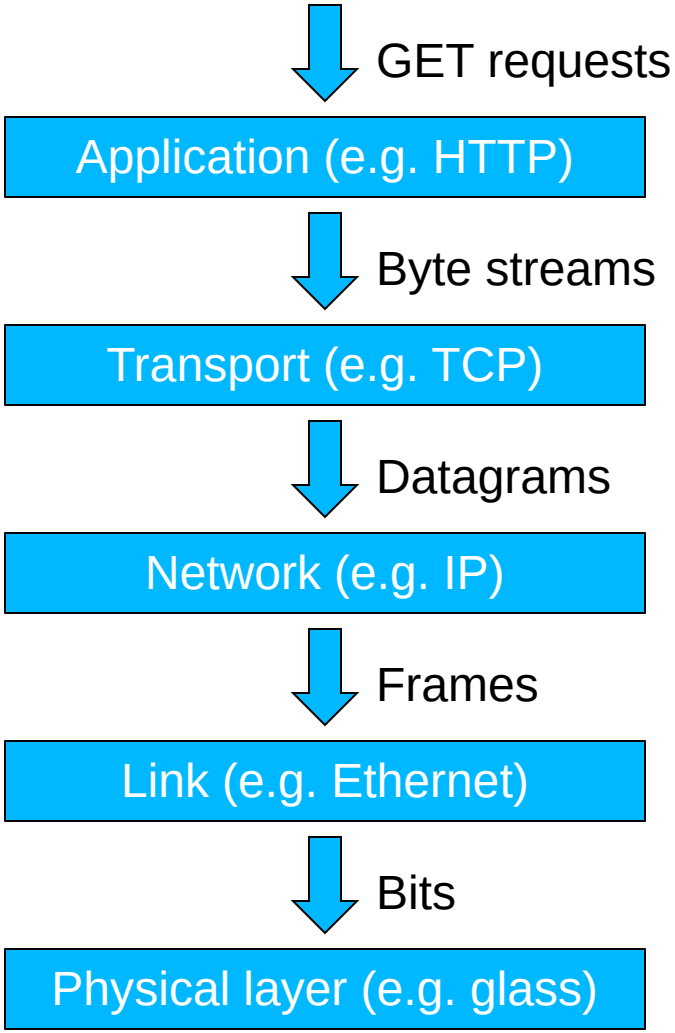
- One system might provide multiple services
  - PostgreSQL provides a variety of different query operations
  - Operating systems host lots of programs
  - CPUs perform integer, FP, load/store, etc.

- Workloads and measurements refer to services
  - Don’t specify a DB workload as a CPU instruction mix
  - Even if want to know which CPU is better for your DB!
  - A good workload should exercise all the relevant services

Services, like systems, are layered



Another example: networks



Level of detail: effort vs. realism

Systems@ETH zürich

Where to get the workload from?

- Only use most frequent request
- Use actual frequency of request types
- Trace-driven workload

For analytical modeling:

- Average resource demands
- Distribution of resource demands

Representativeness

- Arrival rate:
  - Do the requests arrive at realistic times?
  - Web URL requests, DB query types, network packets, etc.
- Resource demands:
  - Does the workload place the same demands on the system as a real application would?
  - CPU load, RAM usage, etc.
- Resource usage profile:
  - Are the resources used in the same sequence and proportions as in a real application?
  - Interactions with other subsystems

Workload selection summary

- Define your SUT, CUS.
- Identify any important external components.
- Choose the correct service interface.
- List the relevant services at the interface.
- Select the level of detail for requests in the workload.
- Make sure it’s representative.
- Make sure it stays representative.
- Decide how much load to apply.
- Ensure it is repeatable.

Workload characterization

The problem

Systems@ETH zürich

- Can we boil down a workload to a small number of quantities that can be used to recreate it?
- How do we find out what quantities matter?
- How can we find out suitable values for them?

What's a user?

Systems@ETH zürich

- A *user* is what makes requests of the SUT…
  - Sometimes a *workload component* or *workload unit*
  - Applications, user sessions, etc.

- Workload *parameters* or *features* characterize the workload
  - Traffic matrix
  - Memory access patterns
  - Transaction load
  - Etc.

Averages

Systems@ETH zürich

- Simplest case: what's the average value?
  - E.g. # requests per second
  - Bytes per packet
  - Transaction size
- But which average?
  - Arithmetic mean
  - Geometric mean
  - Media
  - Mode
  - Harmonic mean

Distributions

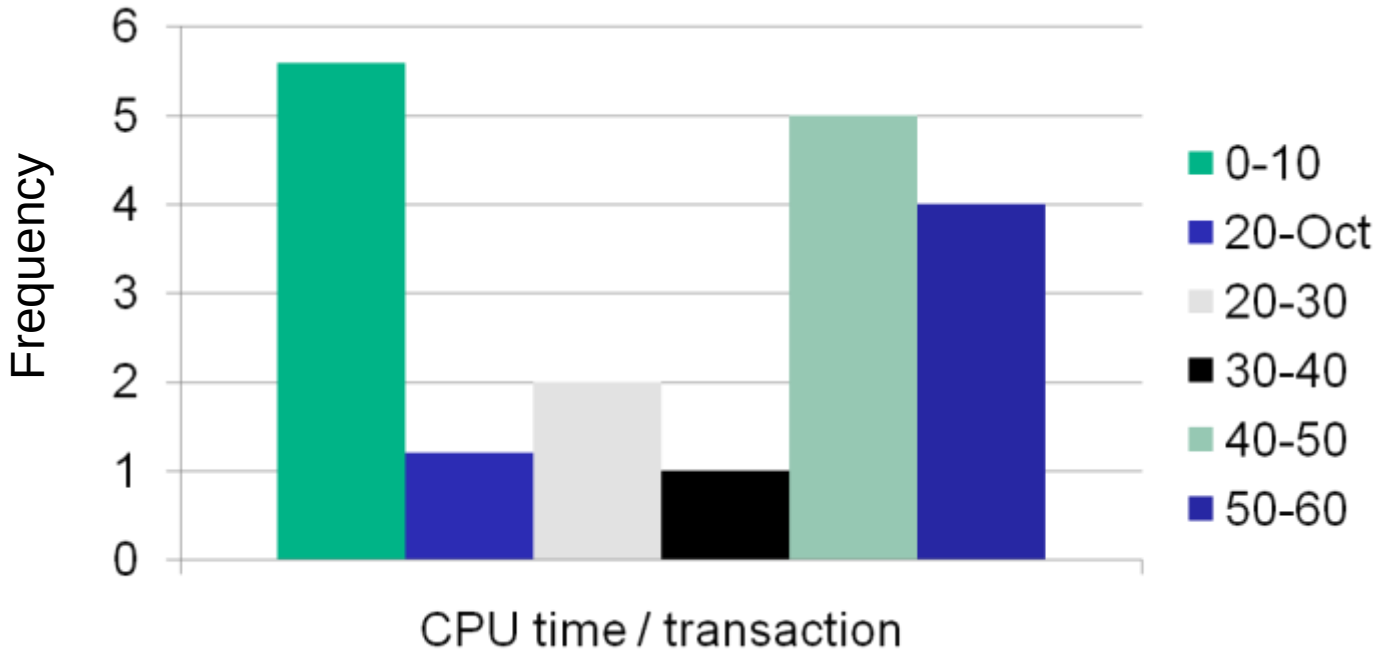
Systems@ETH zürich

- Averages don't convey the variation
  $\Rightarrow$ also specify variance, or standard deviation
- Coefficient of Variation (COV):
  - ratio of standard deviation to mean

- More general: specify the *distribution* of the variable
  - Exponential distribution (e.g. time between failures)
  - Poisson distribution (e.g. number of requests per unit time)
  - Zipf distribution (e.g. popularity rank of items)
  - See later for common distributions, or the book…

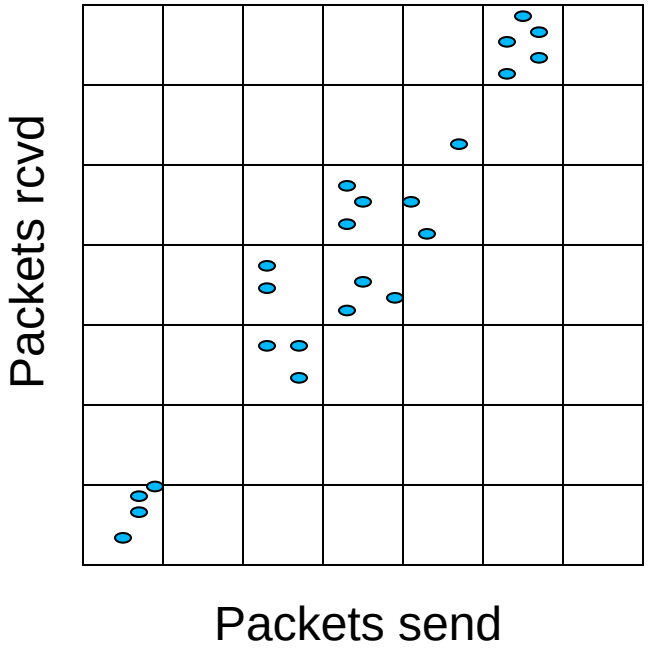
Histograms

- Put possible values into bins with specific frequencies:



Histograms: problems

- Single-parameter histograms ignore *correlations.*
  - E.g. CPU cycles / transaction, I/O ops / transaction
- Beware characterizing workloads by assuming variables are *independent!*
- One solution: multidimensional histograms



Principal component analysis

Systems@ETH zürich

- Often have many potential characterization variables
  - Which are probably highly correlated (dependent)!
  - These constitute a multidimensional *vector space*
- PCA: minimal set of *orthonormal basis vectors* for space
  - I.e. a smaller set of non-correlated variables
  - Characterize workload using these variables
- See book for procedure. Basically:
  - Compute correlation matrix for the variables
  - PC's are obtained from the non-zero (or non-small!) *eigenvectors* of this matrix

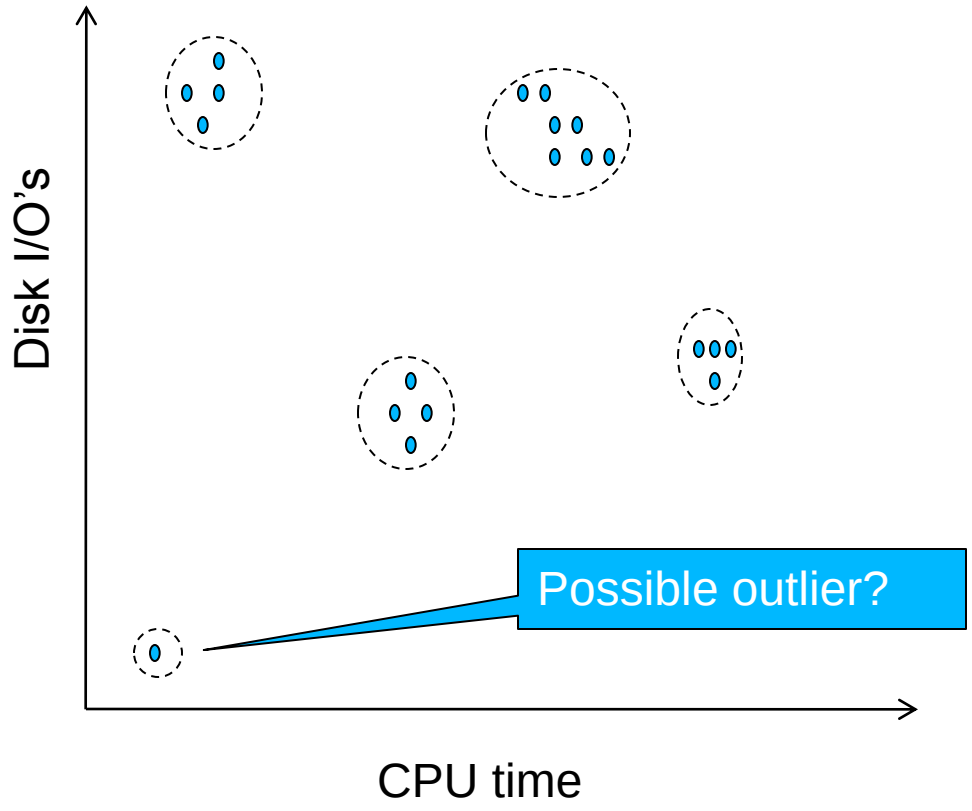
Markov models: sequencing

Systems@ETH zürich

- What if each request depends on previous ones?
- Can characterize using a *Markov Model*:
  - Think of request generator as a state machine
  - State transitions are probabilistic
  - $\Rightarrow$ new state depends only on current state, not the past
  - Represent using a *transition matrix.*

- Generate workload by, e.g. generating random values and running the state machine.

Clustering

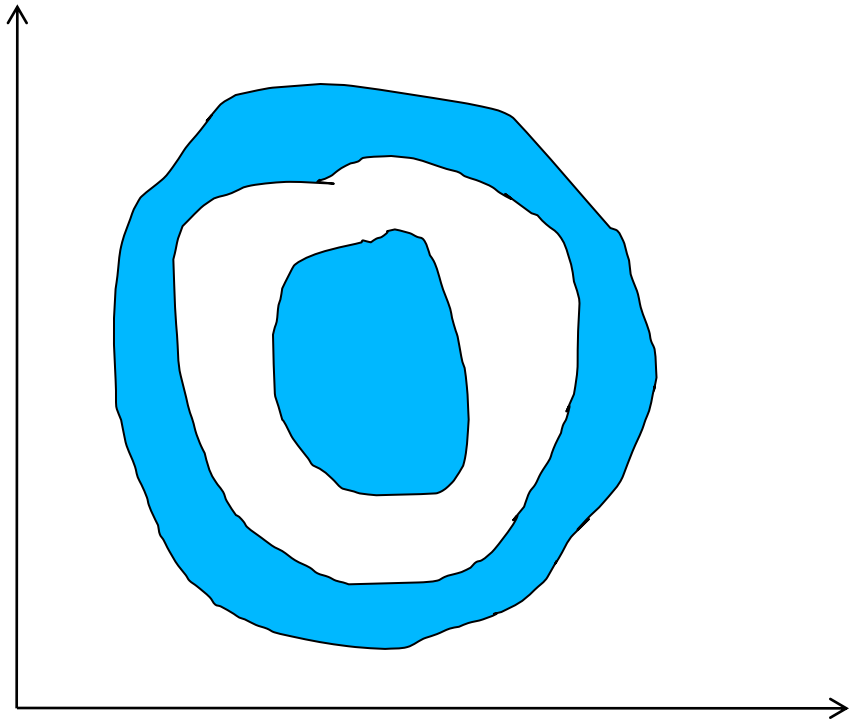


Clustering is hard

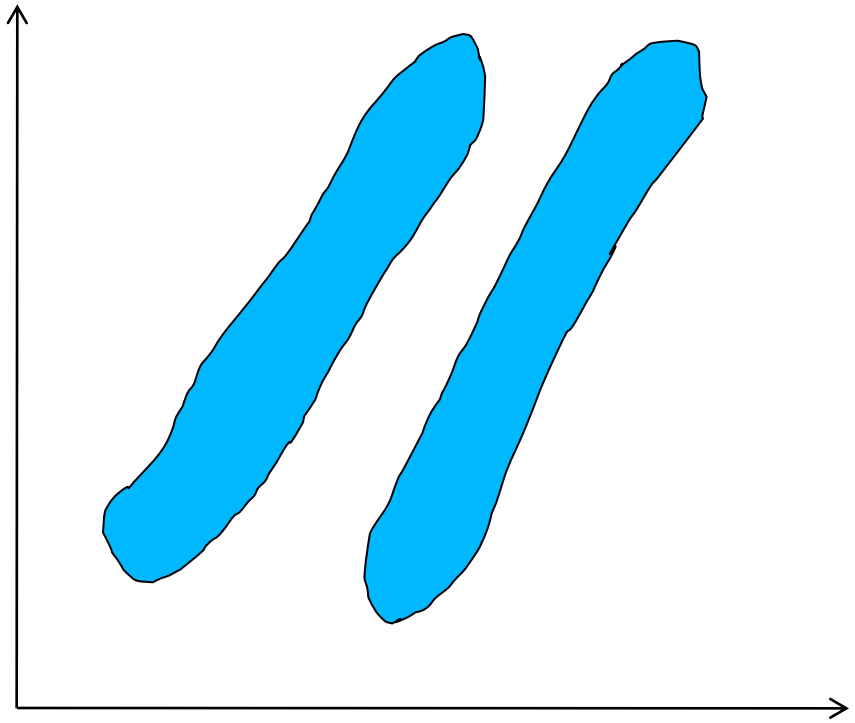
Systems@ETH zürich

- More than just “eyeballing”
  - > 3 variables cannot be eyeballed by humans anyway
- There are many algorithms for cluster analysis
  - They are typically computationally intensive
  - They are typically storage intensive
  - They can go wrong in spectacular ways…
- There is a strong subjective component
  - There may not be an “underlying truth” explaining the clusters
  - What is an outlier, and what is not?

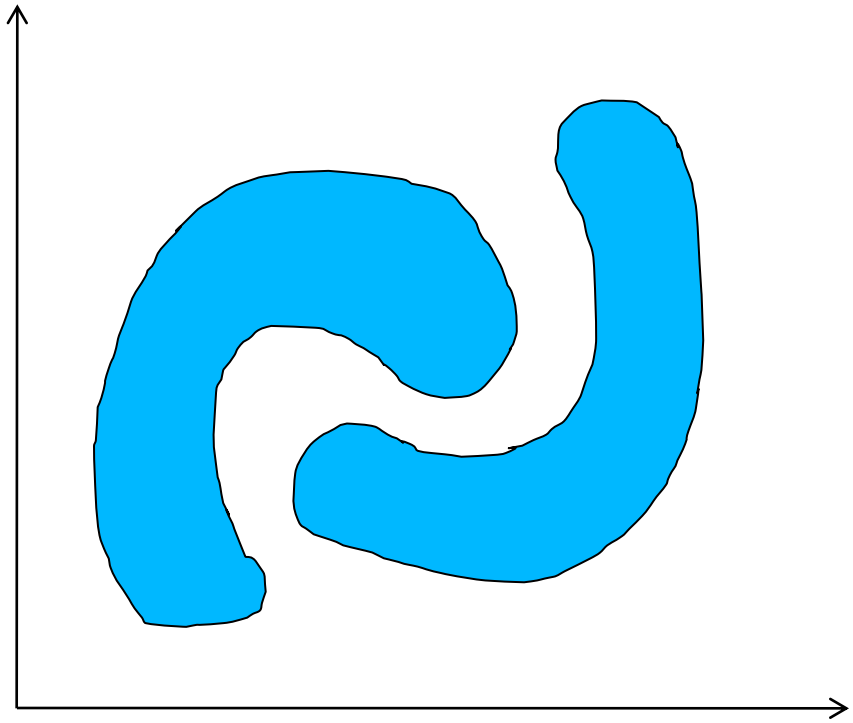
How to confuse cluster algorithms



How to confuse cluster algorithms



How to confuse cluster algorithms



Systems@ETH zürich

Instrumentation and Monitoring

The problem:

Systems@ETH zürich

- How to observe the activities in a system:
  - To obtain a workload trace
  - To obtain results from a performance test

- Almost **everything** you do in systems research involves instrumenting a system (often your own)!
  - Finding performance bottlenecks and optimizing them out
  - Tuning the system parameters
  - Characterizing a workload
  - Finding model parameters or validating models

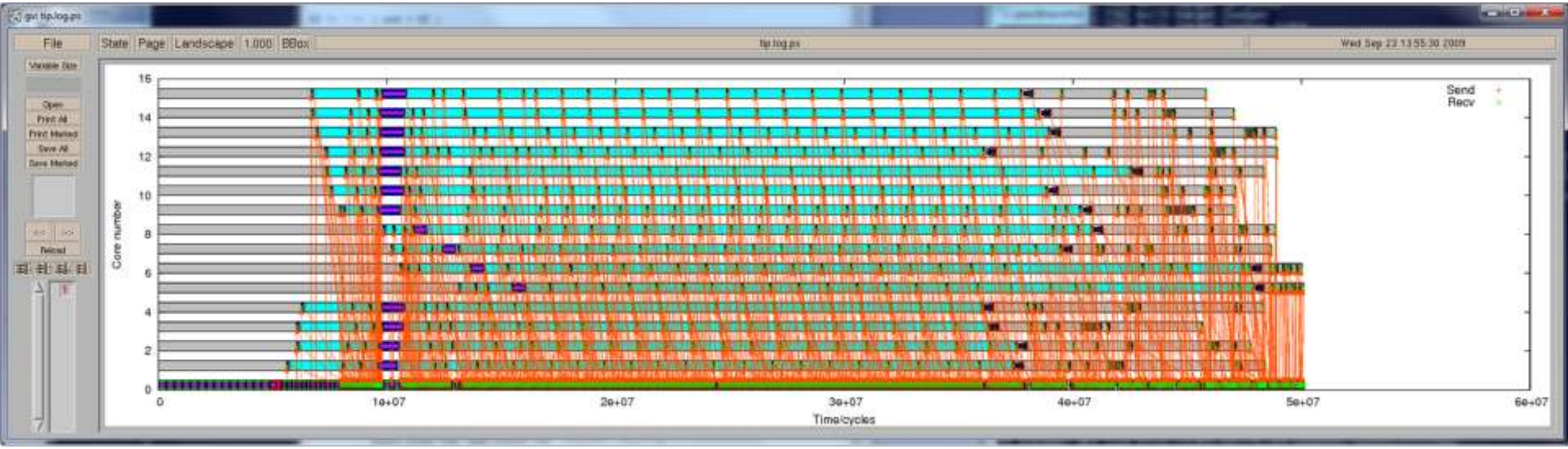
Instrumentation by example

Systems@ETH zürich

- Your lab project measurements are obtained by instrumenting your database client
  - Somewhat “black box” with respect to PostgreSQL.
- Another example for this lecture: Barrelfish
  - “White box”: we put instrumentation inside the OS
- Principles are **general** – these examples are to illustrate.
- Focus: *software* monitoring techniques

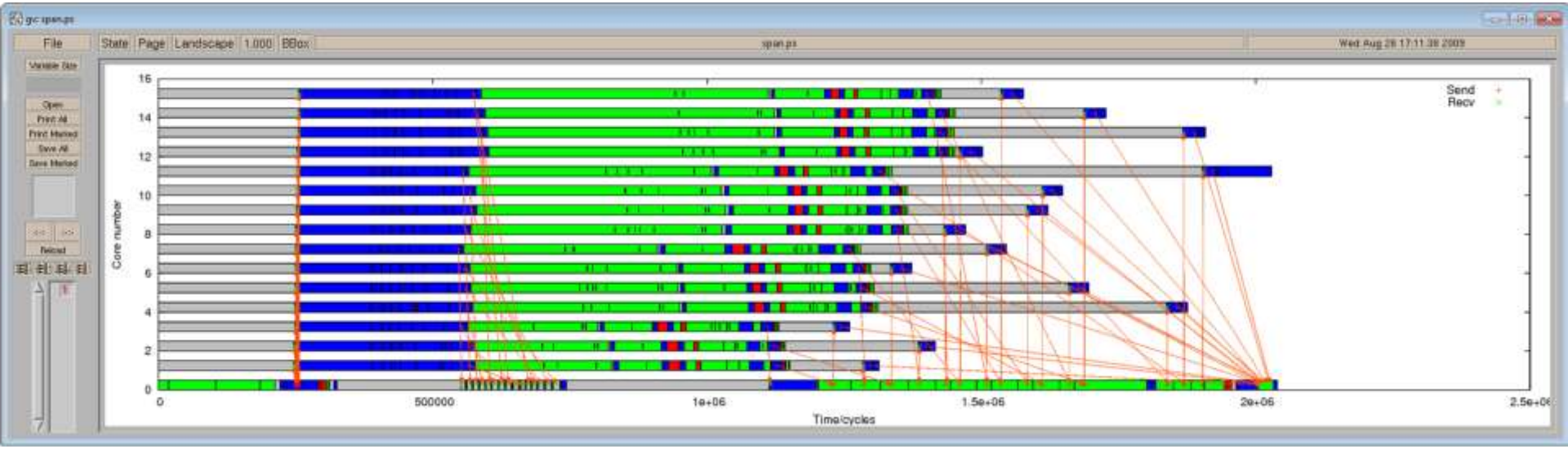
Finding problems

Systems@ETH Zürich



- Trace of messages and context switches in Barrelfish
- x-axis: time (machine cycles)
- y-axis: core number (16-core multiprocessor)
- Color: which process is running
- Problem: starting a program on 16 cores takes **ages.**

Fixing problems



- Reduce messages (in this case to the memory server)
- Partition state
- Fix message polling bugs
- Result: factor of 20 faster!

Instrumentation terminology

- Event: change in system state that is logged
  - Barrelfish example: context switches, message send/recv
- Timestamp: indication of when something happened
  - What's the time? Real time? Logical time?
  - How expensive is reading the time?
    - (probably more than you think…)
  - Barrelfish example: cycle counter (cost ~80 cycles)
- Trace: log of events, usually with *timestamps*
  - Goal is to collect as complete a trace as cheaply as possible
  - Generally kept on disk for off-line analysis
    - Previous slide is updated in realtime!

More terminology

- Overhead: extra resources used for monitoring
  - How cheaply can monitoring be done?
  - How many CPU cycles to record a context switch?
  - How much memory needed to record query plans?
  - Barrelfish example: ~2% of (which is not bad)

- Domain: what is observable?
  - Trace can run for limited time after being *triggered*

More terminology

- Input rate: maximum frequency of observable events
  - You don't want the system to collapse due to event logging
- Resolution: granularity of information observed
  - Try and log everything
  - Various sampling techniques
- Input width: how much information is logged

So, how do you do it?

Systems@ETH zürich

- First, catch your events!

- There are two alternatives:
  1. Modify software to post events
     - Make reporting part of your code
     - Your PostgreSQL clients (should) log queries in line.
  2. Generate regular interrupts to sample the program state
     - How sampling profilers work

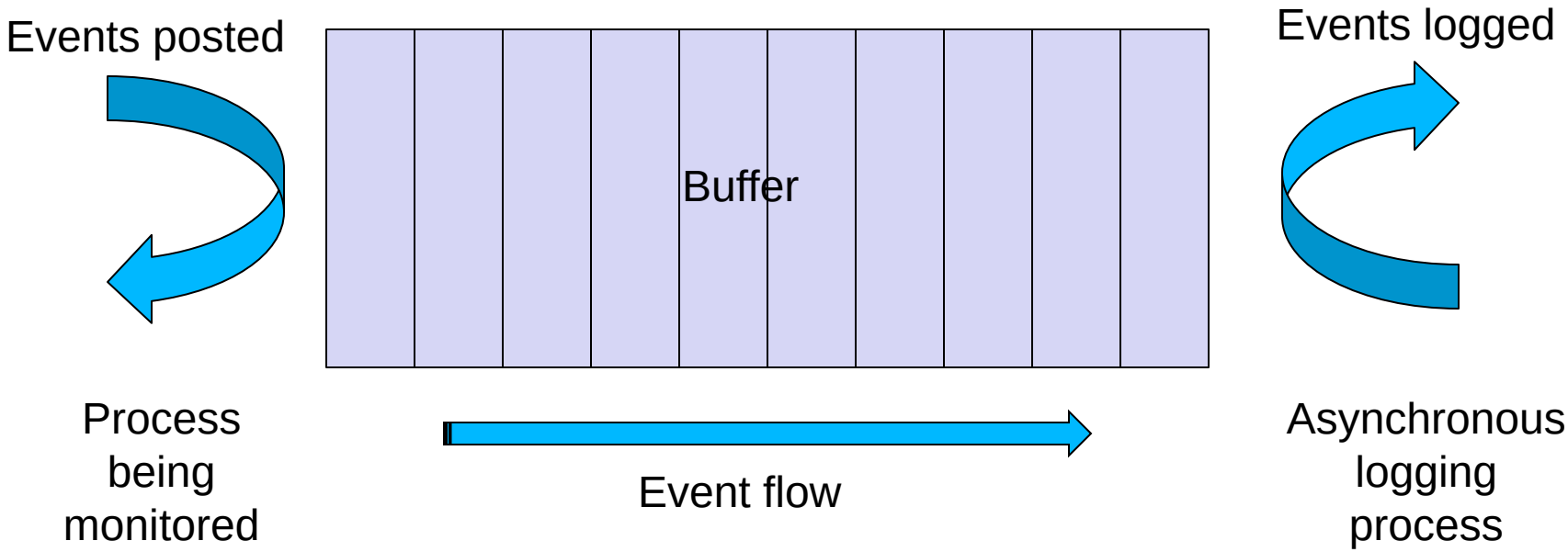
Recording

- Post the event. You need:
  - The time.
    - You'll probably use gettimeofday() or similar
  - A buffer.
    - See later.
  - Encoding, preprocessing, analysis
    - Do you need to fit data into the record size?
    - Can you do data reduction in advance?

The need for trace buffers

- Most of the time, you can't directly log the event
  - Perturbs the system too much, can't wait to continue
- Asynchronous logging:
  - Write the log at a better time
  - Batch the logging operations up
  - Requires a buffer
- In some cases, this comes for free
  - Buffered I/O in C, Java, etc.
- Other cases: you need to implement your own..

Trace buffers



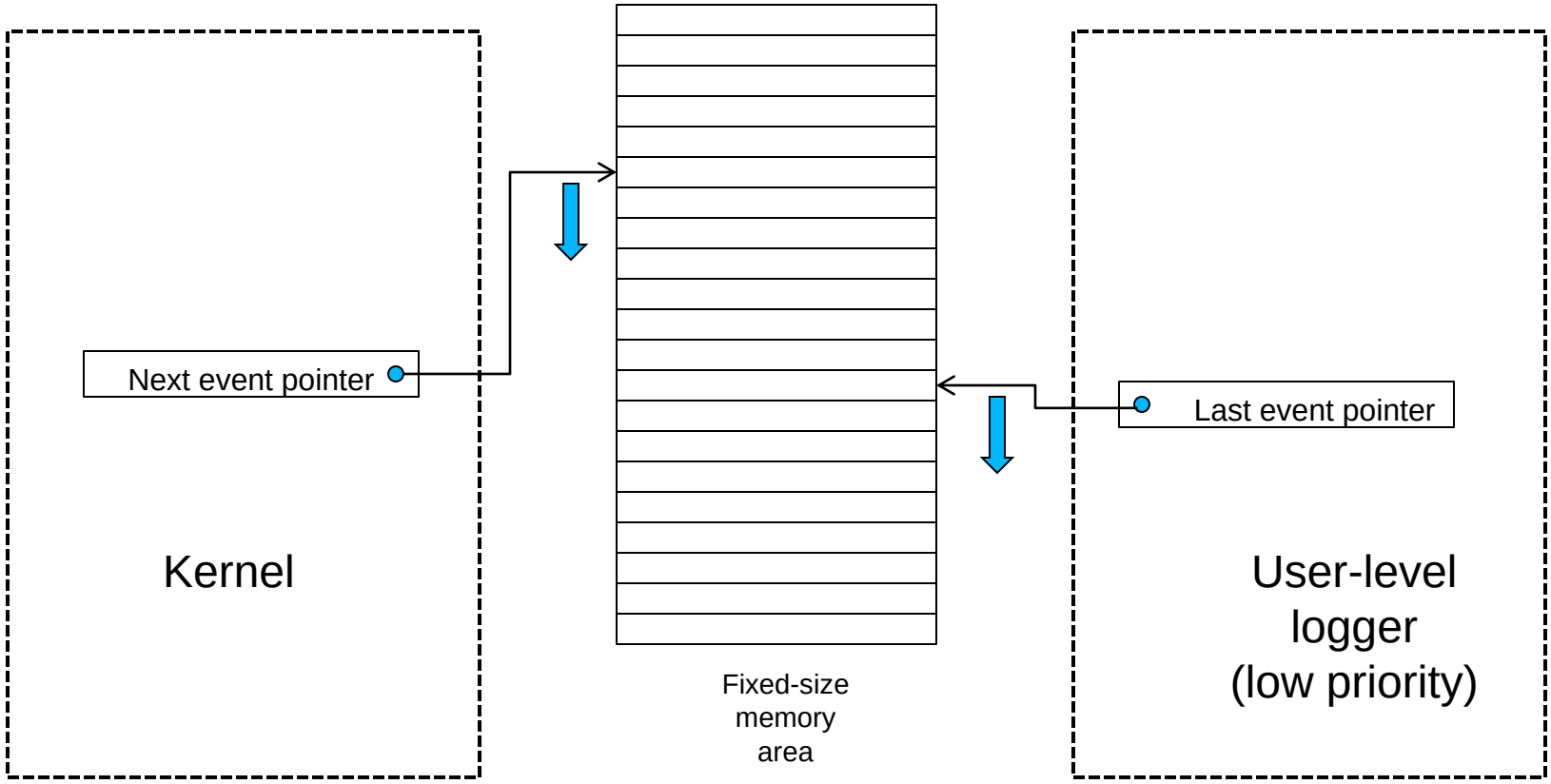
Buffer issues

Systems@ETH zürich

- How big is the buffer?
  - (# events to log) * (space for each event record) ?
- What to do when it overflows?
  - Drop?
  - Overwrite?
  - Stop?
  - Record overflow!
- How to empty the buffer?
  - Asynchronous buffer-writing process (e.g. klogd)
  - Buffer is now shared data structure…

Ring buffers

- Barrelfish uses *ring buffers* (with synchronization)
  - As do many other systems!



Triggers and abnormal events

Systems@ETH zürich

- Turn tracing off and on in response to events themselves
  - Another form of online data reduction…
- Used for
  - Saving buffer space
  - Catching very rare events in context

Scaling

Systems@ETH zürich

- Single buffer is very bad on a multiprocessor
  - Too much contention: overhead dominates!
  - Use multiple (per-core) buffers
  - Aggregate buffers offline (or off-machine)
- Leads to general *distributed monitoring*
  - Typically a pipeline / aggregation tree.
  - Increasing important for analysis, if not performance benchmarking
  - You may find yourself doing this..
- Moral: systems problems in monitoring are often microcosms of systems problems in general