Maximum Performance

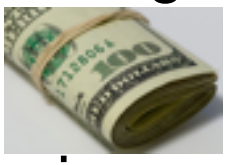
How to get it and how to avoid pitfalls

Christoph Lameter , PhD
cl@linux.com

Performance

Just push a button?

Systems are “optimized” by default for good general performance in all areas.



Optimizations beyond defaults require a sacrifice:

- Money: More expensive systems
- Performance in other areas (interactivity vs. batch)
- Simplicity for complexity
- Maintenance effort
- Highly paid and highly experienced experts for software development and system administration

Today Software APIs limit performance at the high end

The higher level the software API the more overhead which reduces performance. Higher level software APIs are easy to use and allow rapid development of software.

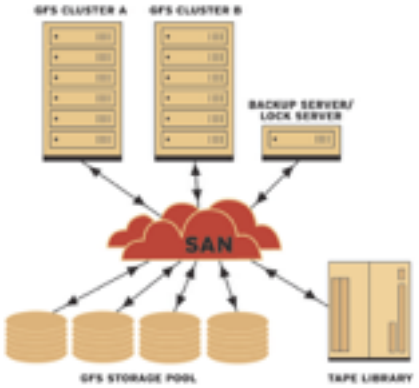
The lower the software API the closer to hardware and the more high performance features of the hardware can be used and the more control is possible over devices etc. The APIs become more difficult to use and require more expertise to use in the right way.

Classic Analysis of Performance Bottlenecks

- Application analysis
  - ➢ *“top” and various diagnostic counters.*
- Process states and their meaning
  - ➢ *Running / D / S*
- Page Faults
  - ➢ *Major faults, minor faults*
- Interrupts and I/O
  - ➢ *Monitor how frequently they occur*
- Latency analysis
  - ➢ *System is optimized for throughput by default.*

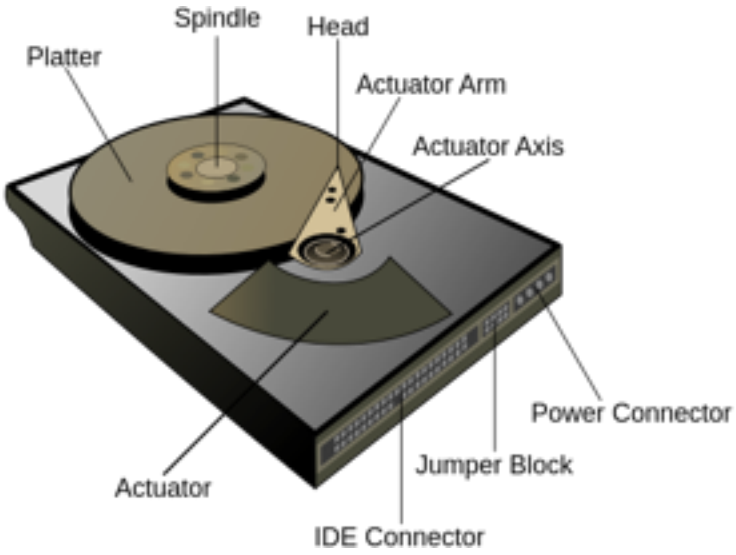
Storage: Optimizing for throughput

- Traditional classic rotational media.
- Today mostly flash based storage
- Large RAID Arrays
- Network storage
- Cloud
- NVRAM/NVDIMM



Classic Rotational Storage Optimization

- Head movements need to be minimized
- Sorting of request by location on disk
- Read a lot where ever the head is
- Send large batches of requests to devices
- Optimize overhead of request submission
- RAID configs to get around the bandwidth limits of devices (~100-300MB/sec) and increase reliability.



However, not for

- **Caching controllers**
- **RAID controllers**
- **Flash storage**
- **Network storage**
- **Cloud**

Storage today is network communication

- There is a system controlling “storage” at the other end.
- This system usually runs Linux as well and caches requests and works them in a device specific way.
- Problem is how to effectively communicate with that system.
- Performance is foremost a networking problem
- And then an issue that the device firmware/software has to deal with

Therefore:

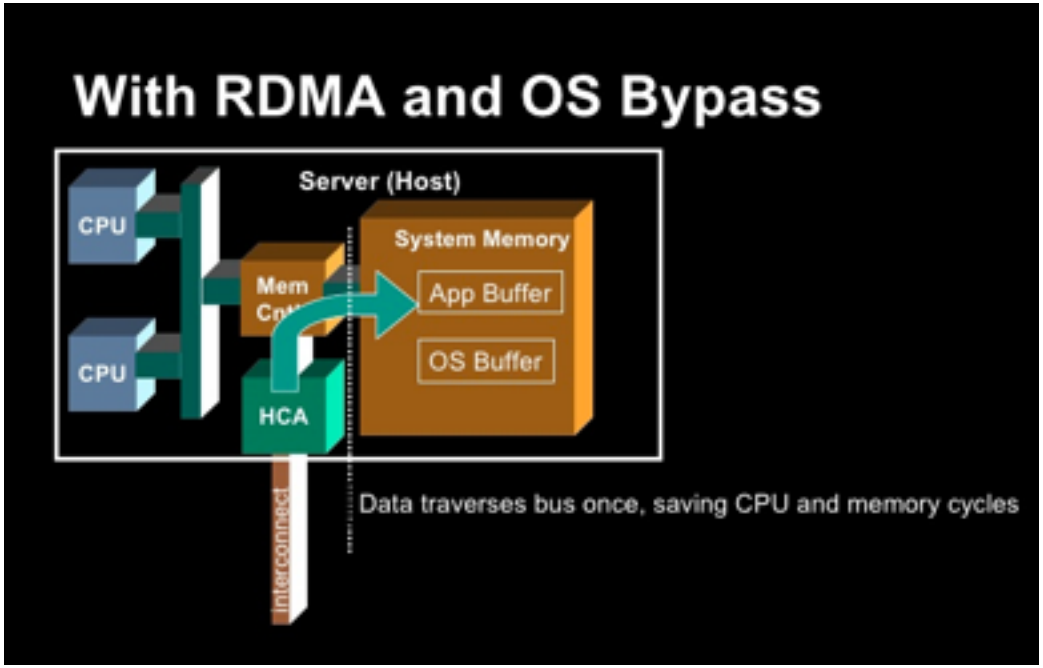
- Block storage layers in the kernel are necessary for legacy reasons. Those limit performance to 1 to 2 Gigabyte per seconds.
- Kernel bypass is possible through network protocols done frequently

Networking: Optimizing for throughput

- Socket API designed for 10M network links
- Works well at 1G. Single thread can handle this.
- Trouble at 10G. Requires multiple threads.
- Higher speed require different APIs and more hardware offload. RDMA? Proprietary offload?
- We are right now introducing 100G networks in the industry. What now?
- Recent work in Redhat by Jesper Dangaard Brouer doubled network stack performance. Able to send ~10mio pps now (idealized test). However, new NICs can send 150 mio pps.

Networking API

- Language specific network access
- Buffered I/O via glibc
- Socket API
- RDMA APIs / Offload APIs
- FPGA (no longer regular coding)
- ASIC
- Analog



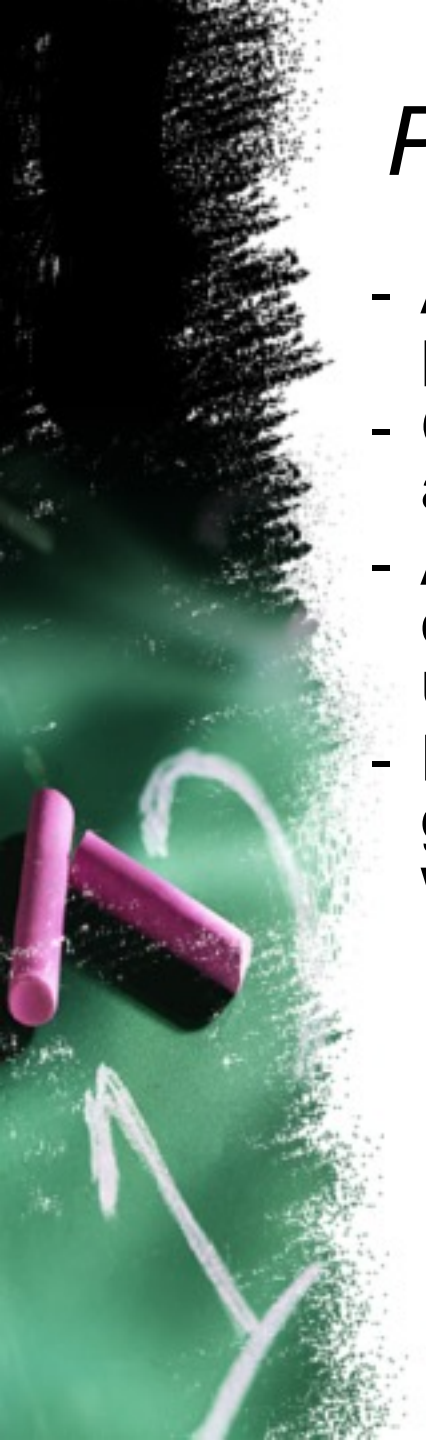
Networking: Optimizing for latency

Default is to optimize for performance!

- *Higher response time than expected*
- *Opportunistic waiting periods in hardware and software.*
- *Power results in constraints on latency*
- *Switches are optimized for throughput*
- *Large packets are evil*

Optimization for memory access

- Memory access depends on effective cpu caching
- TLB misses
- Prefetching
- NUMA
- Increasing complexity of memory access
  - NVDIMM
  - Device mapped memory
  - New levels of caching are continually provided.



Processor Optimizations

- Algorithm depends on the performance of the processor
- OS is seen to be interfering with its maintenance activities.
- Ability to use capabilities of a processor depend on the cache friendly nature of the code. Thus for ultimate performance software needs to be rewritten.
- Moore's law ends. Processor performance cannot grow anymore like in the past. We want the max we can get out of what we paid for.

Floating point throughput optimization

- Parallelism is key here.
- Vector instruction sets for exploiting the parallelism in each core. AVX etc.
- Parallel execution on multiple cores.
- Parallel execution on multiple nodes in a cluster
- Concurrency determines performance. Code execution must be targeted for performance. Thus rewrites by specialists can yield significant improvements
- Dedicated Floating point processors and GPUs
  - ➢ *More Parallel threats*
  - ➢ *Parallel code execution*

Restraining the Operating System

- On multicore system limit Os activity as much as possible to a set of cores on which we cannot get full performance
- The remainder are more or less available fully for the applications.
- Get the OS out of the data path. OS controls but is not directly handling data transfers.
- NOHZ_FULL approach in Linux development upstream
- RDMA APIs of various flavors (DPDK etc etc)

Conclusions

- Optimization changes with the hardware available.
- Devices approach memory speed and therefore existing APIs become problematic.
- Operating system often in the way of the data path. OS needs to exercise control but allow bypass for data transfers.
- For ultimate performance applications need to be redesigned for the hardware they run on.
- Caching is the key for higher performance and they abound in numerous flavors. Mastery of those is required for ultimate performance.