A Brief Lesson in History

[PDF generated June 03 2025. For all recent updates please see the Nutanix Bible releases notes located at https://nutanixbible.com/release_notes.html. Disclaimer: Downloaded PDFs may not always contain the latest information.]

A brief look at the history of infrastructure and what has led us to where we are today.

The Evolution of the Datacenter -

The figure below characterizes the various latencies for specific types of I/O:

Item	Latency	Comments
L1 cache reference	0.5 ns	
L2 cache reference	7 ns	14x L1 cache
DRAM access	100 ns	20x L2 cache, 200x L1 cache
3D XPoint based NVMe SSD read	10,000 of ns (expected)	10 us or 0.01 ms
NAND NVMe SSD R/W	20,000 ns	20 us or 0.02 ms
NAND SATA SSD R/W	50,000-60,000 ns	50-60 us or 0.05-0.06 ms
Read 4K randomly from SSD	150,000 ns	150 us or 0.15 ms
P2P TCP/IP latency (phy to phy)	150,000 ns	150 us or 0.15 ms
P2P TCP/IP latency (vm to vm)	250,000 ns	250 us or 0.25 ms
Read 1MB sequentially from memory	250,000 ns	250 us or 0.25 ms
Round trip within datacenter	500,000 ns	500 us or 0.5 ms
Read 1MB sequentially from SSD	1,000,000 ns	1 ms, 4x memory
Disk seek	10,000,000 ns or 10,000 us	10 ms, 20x datacenter round trip
Read 1MB sequentially from disk	20,000,000 ns or 20,000 us	20 ms, 80x memory, 20x SSD
Send packet CA -> Netherlands -> CA	150,000,000 ns	150 ms

(credit: Jeff Dean, https://gist.github.com/jboner/2841832)

The table above shows that the CPU can access its caches at anywhere from ~0.5-7ns (L1 vs. L2). For main memory, these accesses occur at ~100ns, whereas a local 4K SSD read is ~150,000ns or 0.15ms.

If we take a typical enterprise-class SSD (in this case the Intel S3700 - SPEC), this device is capable of the following:

- Random I/O performance:
  - Random 4K Reads: Up to 75,000 IOPS
  - Random 4K Writes: Up to 36,000 IOPS

- Sequential bandwidth:
  - Sustained Sequential Read: Up to 500MB/s
  - Sustained Sequential Write: Up to 460MB/s

- Latency:
  - Read: 50us
  - Write: 65us

Looking at the Bandwidth

For traditional storage, there are a few main types of media for I/O:

- Fiber Channel (FC)
    - 4-, 8-, 16- and 32-Gb

- Ethernet (including FCoE)
    - 1-, 10-Gb, (40-Gb IB), etc.

For the calculation below, we are using the 500MB/s Read and 460MB/s Write BW available from the Intel S3700.

The calculation is done as follows:

```
numSSD = ROUNDUP((numConnections * connBW (in GB/s))/ ssdBW (R or W))
```

NOTE: Numbers were rounded up as a partial SSD isn't possible. This also does not account for the necessary CPU required to handle all of the I/O and assumes unlimited controller CPU power.

Network BW		**SSDs required to saturate network BW**	
Controller Connectivity	**Available Network BW**	**Read I/O**	**Write I/O**
Dual 4Gb FC	8Gb == 1GB	2	3
Dual 8Gb FC	16Gb == 2GB	4	5
Dual 16Gb FC	32Gb == 4GB	8	9
Dual 32Gb FC	64Gb == 8GB	16	19
Dual 1Gb ETH	2Gb == 0.25GB	1	1
Dual 10Gb ETH	20Gb == 2.5GB	5	6

As the table shows, if you wanted to leverage the theoretical maximum performance an SSD could offer, the network can become a bottleneck with anywhere from 1 to 9 SSDs depending on the type of networking leveraged

The Impact to Memory Latency

Typical main memory latency is ~100ns (will vary), we can perform the following calculations:

- Local memory read latency = 100ns + [OS / hypervisor overhead]
- Network memory read latency = 100ns + NW RTT latency + [2 x OS / hypervisor overhead]

If we assume a typical network RTT is ~0.5ms (will vary by switch vendor) which is ~500,000ns that would come down to:

- Network memory read latency = 100ns + 500,000ns + [2 x OS / hypervisor overhead]

If we theoretically assume a very fast network with a 10,000ns RTT:

- Network memory read latency = 100ns + 10,000ns + [2 x OS / hypervisor overhead]

What that means is even with a theoretically fast network, there is a 10,000% overhead when compared to a non-network memory access. With a slow network this can be upwards of a 500,000% latency overhead.

In order to alleviate this overhead, server side caching technologies are introduced.

User vs. Kernel Space

One frequently debated topic is the argument between doing things in kernel vs. in user-space. Here I'll explain what each is and their respective pros/cons.

Any operating system (OS) has two core areas of execution:

- Kernel space
    - The most priviliged part of the OS
    - Handles scheduling, memory management, etc.
    - Contains the physical device drivers and handles hardware interaction

- User space
    - "Everything else"
    - This is where most applications and processes live
    - Protected memory and execution

These two spaces work in conjunction for the OS to operate. Now before moving on let's define a few key items:

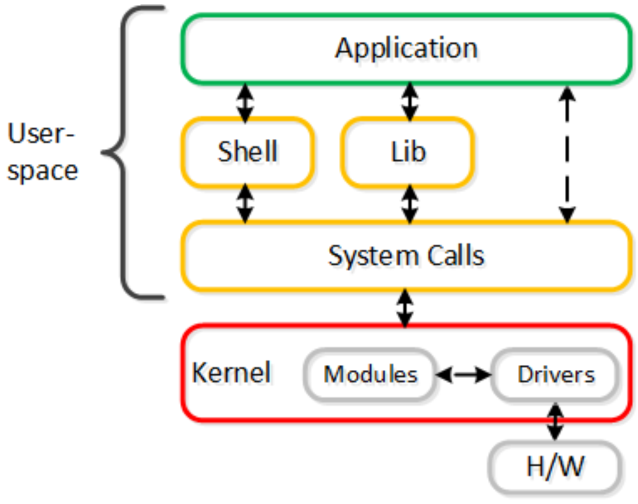
- System call
    - A.k.a. kernel call, a request made via interrupt (more here later) from an active process that something be done by the kernel

- Context switch
    - Shifting the execution from the process to the kernel and vice-versa

For example, take the following use-case of a simple app writing some data to disk. In this the following would take place:

1. App wants to write data to disk
2. Invokes a system call
3. Context switch to kernel
4. Kernel copies data
5. Executes write to disk via driver

The following shows a sample of these interactions:



User and Kernel Space Interaction

Is one better than the other? In reality there are pros and cons for each:

- User space
    - Very flexible
    - Isolated failure domains (process)
    - *Can be* inefficient
        - Context switches cost time(~1,000ns)

- Kernel space
    - Very rigid
    - Large failure domain

    - *Can be* efficient
        - Reduces context switches

Polling vs. Interrupts

Another core component is how the interaction between the two is handled. There are two key types of interaction:

- Polling
    - Constantly "poll" e.g. consistently ask for something
    - Examples: Mouse, monitor refresh rate, etc.
    - Requires constant CPU, but much lower latency
    - Eliminates expense of kernel interrupt handler
        - Removes context switch

- Interrupt
    - "Excuse me, I need foo"
    - Example: Raising hand to ask for something
    - Can be more "CPU efficient", but not necessarily
    - Typically much higher latency

The Move to User Space / Polling

As devices have become far faster (e.g. NVMe, Intel Optane, pMEM), the kernel and device interaction has become a bottleneck. To eliminate these bottlenecks, a lot of vendors are moving things **out of the kernel** to user space with polling and seeing much better results.

A great example of this are the Intel Storage Performance Development Kit (SPDK) and Data Plane Development Kit (DPDK). These projects are geared at maximizing the performance and reducing latency as much as possible, and have shown great success.

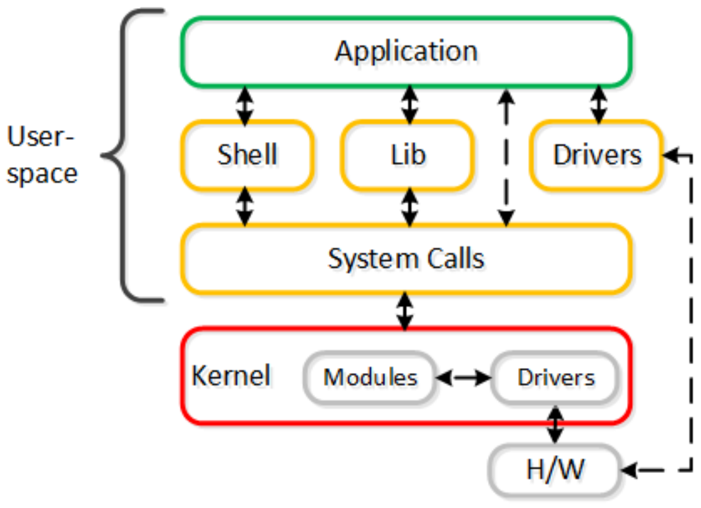
This shift is composed of two core changes:

1. Moving device drivers to user space (instead of kernel)
2. Using polling (instead of interrupts)

This enables far superior performance when compared to the kernel based predecessors, as it eliminates:

- Expensive system calls and the interrupt handler
- Data copies
- Context switches

The following shows the device interaction using user space drivers:



User Space and Polling Interaction

In fact, a piece of software Nutanix had developed for their AHV product (vhost-user-scsi), is actually being used by Intel for their SPDK project.

Web-Scale

web·scale - /web ' skāl/ - noun - computing architecture

a new architectural approach to infrastructure and computing.

This section will present some of the core concepts behind "Web-scale" infrastructure and why we leverage them. Before we get started, it should be stated that the Web-scale doesn't mean you need to be "web-scale" (e.g. Google, Facebook, or Microsoft). These constructs are applicable and beneficial at any scale (3-nodes or thousands of nodes).

Historical challenges included:

- Complexity, complexity, complexity
- Desire for incremental based growth
- The need to be agile

There are a few key constructs used when talking about "Web-scale" infrastructure:

- Hyper-convergence
- Software defined intelligence
- Distributed autonomous systems
- Incremental and linear scale out

Other related items:

- API-based automation and rich analytics
- Security as a core tenant
- Self-healing

The following sections will provide a technical perspective on what they actually mean.

Hyper-Convergence

There are differing opinions on what hyper-convergence actually is. It also varies based on the scope of components (e.g. virtualization, networking, etc.). However, the core concept comes down to the following: natively combining two or more components into a single unit. 'Natively' is the key word here. In order to be the most effective, the components must be natively integrated and not just bundled together. In the case of Nutanix, we natively converge compute + storage to form a single node used in our appliance. For others, this might be converging storage with the network, etc.

What it really means:

- Natively integrating two or more components into a single unit which can be easily scaled

Benefits include:

- Single unit to scale
- Localized I/O
- Eliminates traditional compute / storage silos by converging them

Software-Defined Intelligence

Software-defined intelligence is taking the core logic from normally proprietary or specialized hardware (e.g. ASIC / FPGA) and doing it in software on commodity hardware. For Nutanix, we take the traditional storage logic (e.g. RAID, deduplication, compression, etc.) and put that into software that runs in each of the Nutanix Controller VMs (CVM) on standard hardware.

Supported Architectures

Nutanix currently supports both x86 and IBM POWER architectures.

What it really means:

- Pulling key logic from hardware and doing it in software on commodity hardware

Benefits include:

- Rapid release cycles
- Elimination of proprietary hardware reliance
- Utilization of commodity hardware for better economics
- Lifespan investment protection

To elaborate on the last point: old hardware can run the latest and greatest software. This means that a piece of hardware years into its depreciation cycle can run the latest shipping software and be feature parity with new deployments shipping from the factory.

Distributed Autonomous Systems

Distributed autonomous systems involve moving away from the traditional concept of having a single unit responsible for doing something and distributing that role among all nodes within the cluster. You can think of this as creating a purely distributed system. Traditionally, vendors have assumed that hardware will be reliable, which, in most cases can be true. However, core to distributed systems is the idea that hardware will eventually fail and handling that fault in an elegant and non-disruptive way is key.

These distributed systems are designed to accommodate and remediate failure, to form something that is self-healing and autonomous. In the event of a component failure, the system will transparently handle and remediate the failure, continuing to operate as expected. Alerting will make the user aware, but rather than being a critical time-sensitive item, any remediation (e.g. replace a failed node) can be done on the admin's schedule. Another way to put it is fail in-place (rebuild without replace) For items where a "leader" is needed, an election process is utilized. In the event this leader fails a new leader is elected. To distribute the processing of tasks MapReduce concepts are leveraged.

What it really means:

- Distributing roles and responsibilities to all nodes within the system
- Utilizing concepts like MapReduce to perform distributed processing of tasks
- Using an election process in the case where a "leader" is needed

Benefits include:

- Eliminates any single points of failure (SPOF)
- Distributes workload to eliminate any bottlenecks

Incremental and linear scale out

Incremental and linear scale out relates to the ability to start with a certain set of resources and as needed scale them out while linearly increasing the performance of the system. All of the constructs mentioned above are critical enablers in making this a reality. For example, traditionally you'd have 3-layers of components for running virtual workloads: servers, storage, and network – all of which are scaled independently. As an example, when you scale out the number of servers you're not scaling out your storage performance. With a hyper-converged platform like Nutanix, when you scale out with new node(s) you're scaling out:

- The number of hypervisor / compute nodes
- The number of storage controllers
- The compute and storage performance / capacity
- The number of nodes participating in cluster wide operations

What it really means:

- The ability to incrementally scale storage / compute with linear increases to performance / ability

Benefits include:

- The ability to start small and scale
- Uniform and consistent performance at any scale

Making Sense of It All

In summary:

1. Inefficient compute utilization led to the move to virtualization
2. Features including vMotion, HA, and DRS led to the requirement of centralized storage
3. VM sprawl led to increased load and contention on storage
4. SSDs came in to alleviate the issues but changed the bottleneck to the network / controllers
5. Cache / memory accesses over the network face large overheads, minimizing their benefits
6. Array configuration complexity still remains the same
7. Server side caches were introduced to alleviate the load on the array / impact of the network, however introduces another component to the solution
8. Locality helps alleviate the bottlenecks / overheads traditionally faced when going over the network
9. Shifts the focus from infrastructure to ease of management and simplifying the stack
10. The birth of the Web-Scale world!