40 Gigabytes per second through the Linux page cache

By Christoph Lameter, October 29, 2019

http://gentwo.org/christoph/40G-pagecache

cl@linux.com Open Source Summit Europe Twitter: @qant

Page cache performance issues

In our computing environment we continue to see issues with page cache I/O. Most of software does not use special access modes but it written for regular I/O and with that we are limited to about 2-3Gbyte per second from a single thread.

There are ways to work around the performance issues using direct I/O or huge pages but that usually requires application modifications.

How to Scale Regular I/O on Linux when running on Intel processors

40GB/sec means updating the status of 10 Million 4k pages per second. I/O is limited by the Kernel ability to update the status of 10 million cache lines while doing I/O.

A Memory Management Perspective

- Given the speed of modern NVME drives it is the *OS overhead* that determines **Page Cache** performance.
- Memory latencies, cross node memory accesses and cache hotness are key for performance.
- We need to optimize the computational speed of handling 10 million 4K pages per second.
- Use NUMA features of the system to segment the system into portions that run in parallel
- Scale Page cache performance

A unique Intel legacy problem with a 4K page size

- 4k is tiny given modern data transfers. 64K is even available on ARM. Not so on Intel
- 64K page size cuts down the number of pages to be handled by the OS from 10 Million to 600000 and significantly improves performance. In our experience this problems does not exist on other platforms.
- Intel supports 2M page sizes and there are efforts to allow mixed pages of 2M and 4K in the Linux Kernel but it may take years to complete that work.
- Doing so is complex and possibly runs against problems of fragmentation that have so far been considered unsolvable.
- With 2M pagesize the number of pages to handle would drop to ~20000.
- (1G Pagesize is also possible on x86 which would let this go down to 40 operations. 1G page size is currently not well supported)

Direct I/O and Huge Pages

- Direct I/O bypasses some of the OS overhead by reducing 4k page processing.
- But huge pages are not put in the page cache. Therefore the contents of files are not shared between processes. Data may have to be reread.
- Direct I/O requires application changes.
- Huge Pages ensure contiguous memory segments of 2M each but can only be used with Direct I/O.

The Hardware the **Intel RULER**

We got a system from Intel called the *Ruler* system with 32 NVME drives. With SPDK this can run at 50-56 GB/second ("up to 64GB/s").

This presentation is about how to get close to that using regular I/O through the page cache with normal applications

*Supermicro **1U** "Ruler" Server,* 768G RAM, 100G Ethernet
2 NUMA Sockets *Skylake-6148*, 20 Cores
32 NVME drives 8TB each for a total of 256G
(Future system will have 32TB for **1PB in 1U**)
32 PCIe Lanes per processor going to a PCIe switch

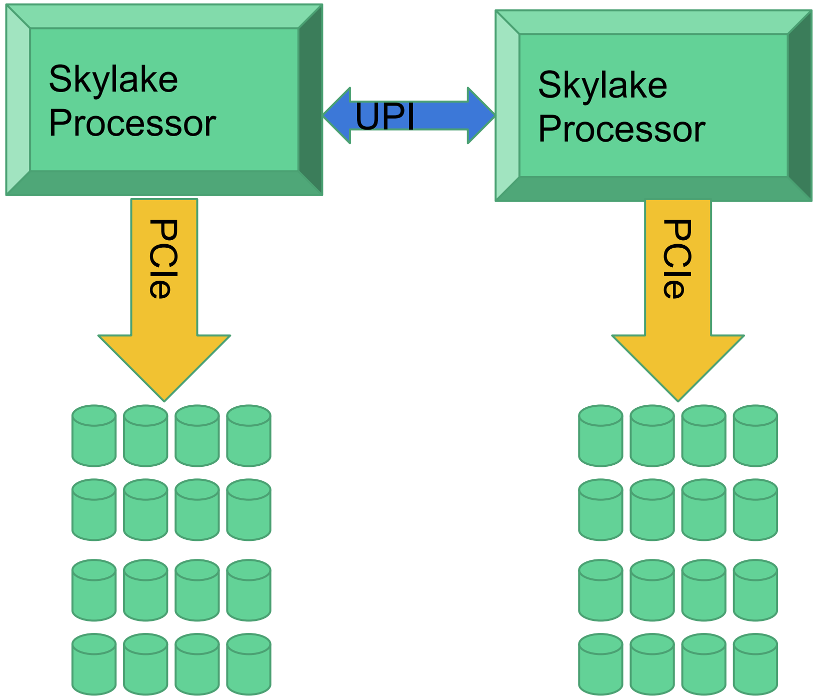
The Storage Architecture

2 Sockets with 384 Gbyte of memory each and a 20 Core processor.

UPI cross connect between the two sockets

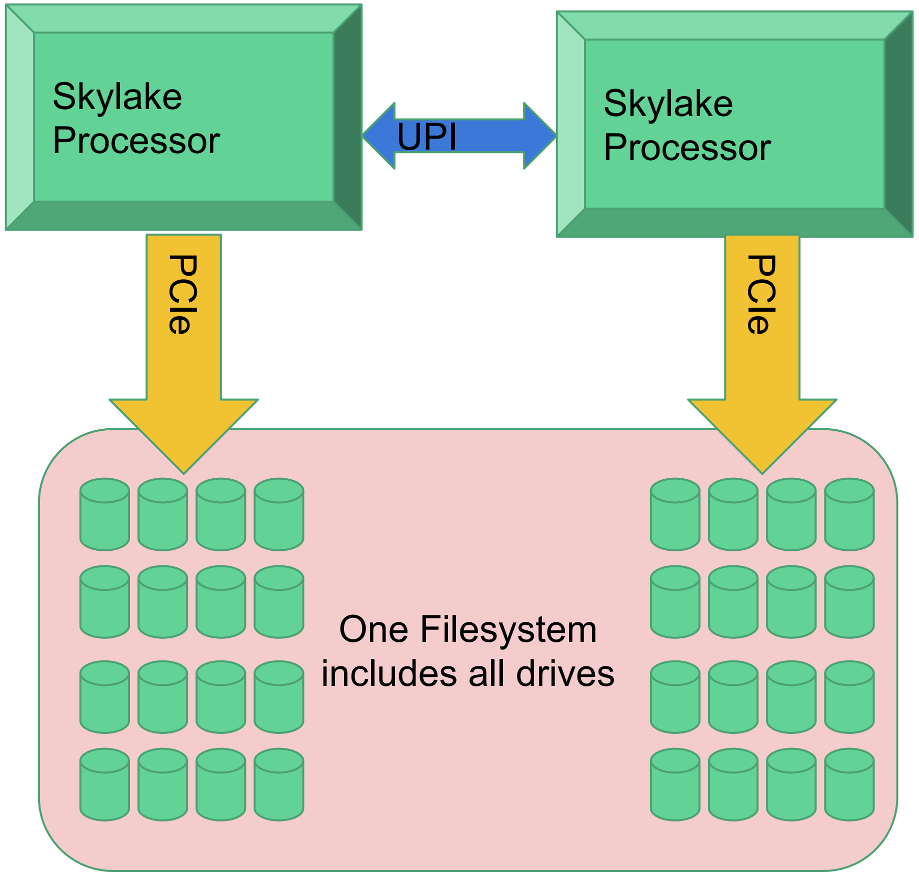
16 NVME drives attached via a special PCIe switch to each processor.

The “Ruler” architecture contains cross PCIe lanes between both sockets too but as far as I can tell they are useless and not really supported by the Operating System. They will support PCIe multipathing at some point I guess.



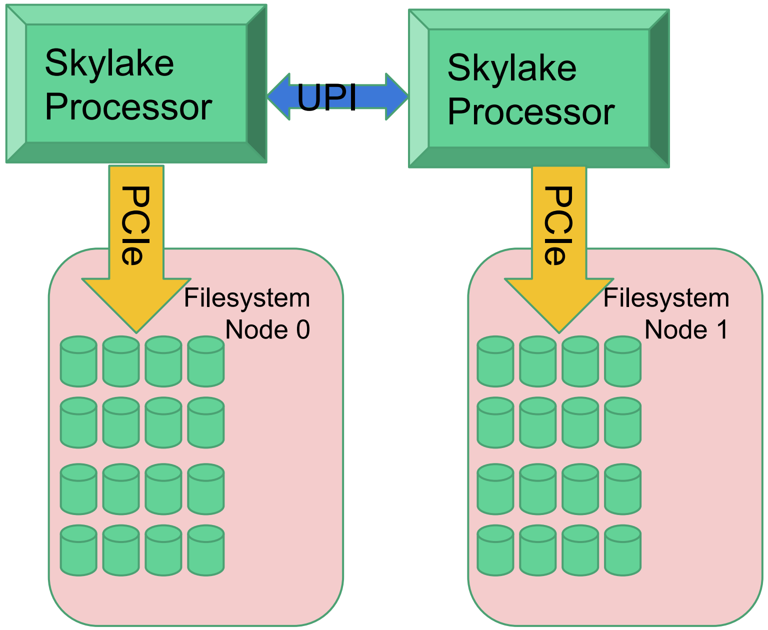
Test Setup “No NUMA”

- NUMA localities not configured
- The Linux Kernel determines the placements of threads and I/O activities.
- There is a single volume striped between both groups of NVME drives.
- Simplest configuration.
- Lock contention across the sockets which causes latencies via the UPI link.



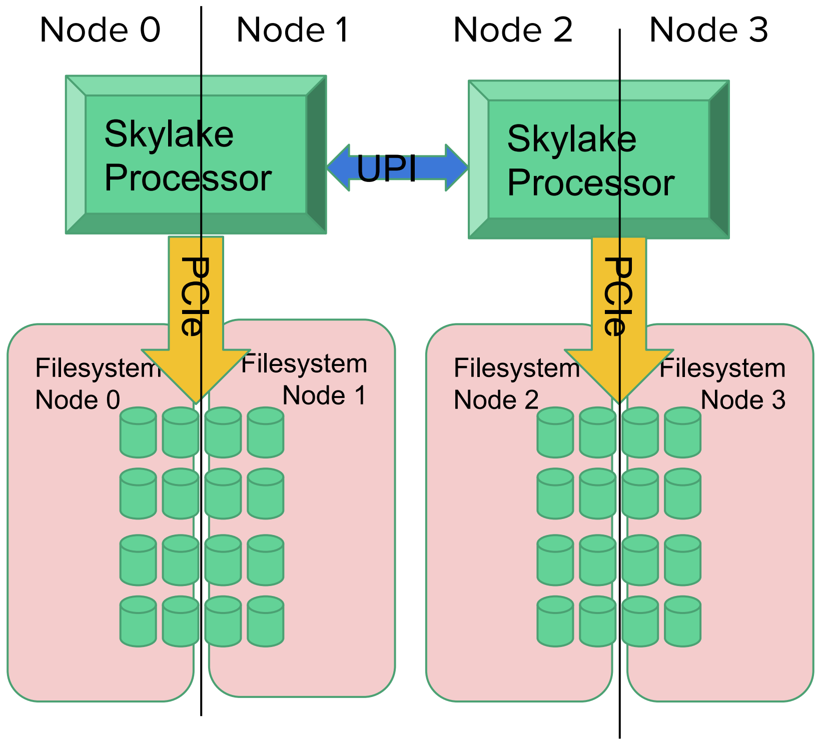
NUMA Setup with 2 Nodes

- Threads and I/O bound to a NUMA node
- The Linux Kernel is restricted to scheduling within a socket.
- Two volumes are created for each node so that each node can do local I/O without serialization with the other node.
- A straightforward NUMA configuration parallelizing I/O



Sub NUMA Clustering Setup with 4 Nodes

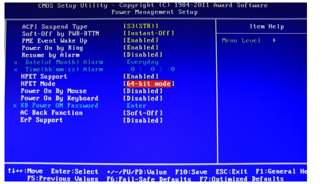
- BIOS configurations in Skylake processors allow splitting the processor into 2 slices.
- Each Skylake processor is managed as 2 NUMA nodes for a total of 4 NUMA nodes in the system.
- Four volumes are created for each node. I/O for each node flows through dedicated PCIe Lanes.
- The most advanced parallelizing configuration possible.



BIOS issues with SNC

SNC splits the Skylake processor into two halves each with their own memory controller and PCIe links.

SNC is a feature provided by Intel and so there was a setting in the BIOS configuration but apparently this was not tested by the OEM.

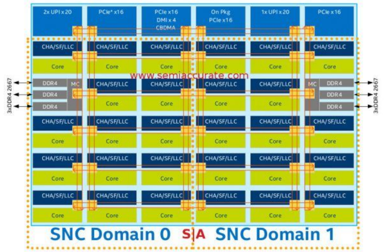


```
          CMOS Setup Utility - Copyright (C) 1984-2011 Award Software
                            Power Management Setup

   ACPI Suspend Type          [S3(STR)]             Item Help
   Soft-Off by PWR-BTTN       [Instant-Off]
   PME Event Wake Up          [Enabled]        Menu Level  ▶
   Power On by Ring           [Enabled]
   Resume by Alarm            [Disabled]
 x  Date(of Month) Alarm       Everyday
 x  Time(hh:mm:ss) Alarm        0 :  0 :  0
   HPET Support               [Enabled]
   HPET Mode                  [64-bit mode]
   Power On By Mouse          [Disabled]
   Power On By Keyboard       [Disabled]
 x KB Power ON Password        Enter
   AC Back Function           [Soft-Off]
   ErP Support                [Disabled]

↑↓→←:Move  Enter:Select  +/-/PU/PD:Value  F10:Save  ESC:Exit  F1:General He
    F5:Previous Values  F6:Fail-Safe Defaults  F7:Optimized Defaults
```

- BIOS crash when selected
- NUMA localities incorrect when it booted (we now have 4 NUMA nodes not 2!)
- Direct contact with BIOS engineers of OEM to fix bugs was required before it became usable.



After a couple of BIOS updates this actually worked right.

Tests were done using the "fio" tool

We are mainly interested in write speed so we only do sequential write tests.

These are 4GB sequential writes.

Scripts etc used are available from http://gentwo.org/christoph/40G-page cache

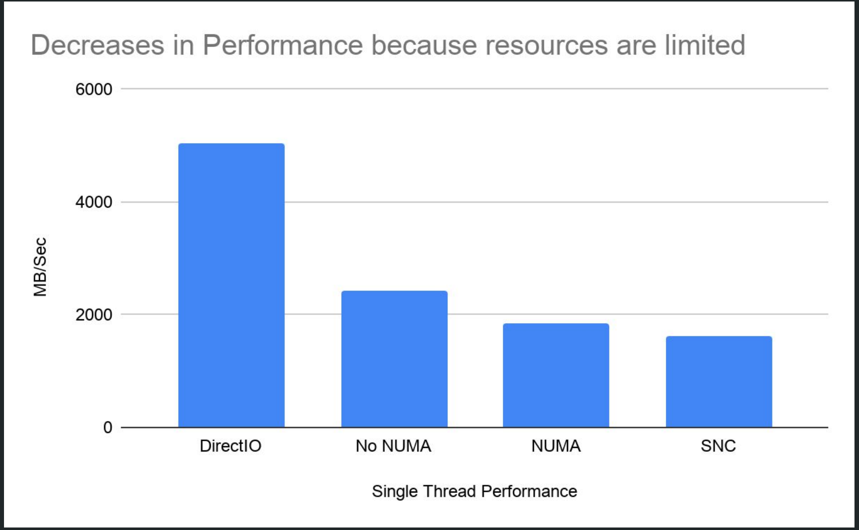
Note that segmenting the system causes issues:

- Applications need to access the "local" filesystem. Storage is localized like memory. This may require application changes.
- The "fio" tool supports binding threads and the files created to nodes and those options were used.
- Sharing of data between the "segments" must be carefully done to avoid performance loss.

Single Thread Performance

Segmenting the system through NUMA and SNC reduces the resources available to a single core

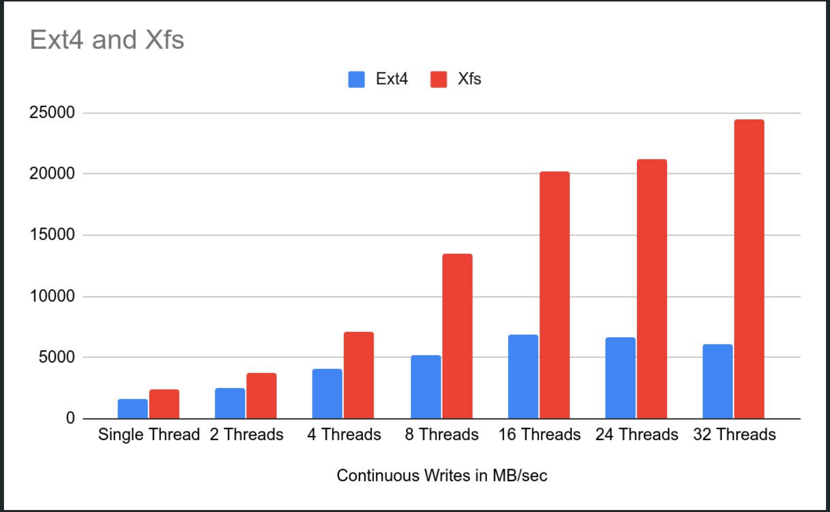
Segmentation requires additional parallel tasks to compensate for the performance loss



Ext4 vs XFS Performance

Ext4 is not geared for multicore performance as much. Throughput sinks above 16 threads

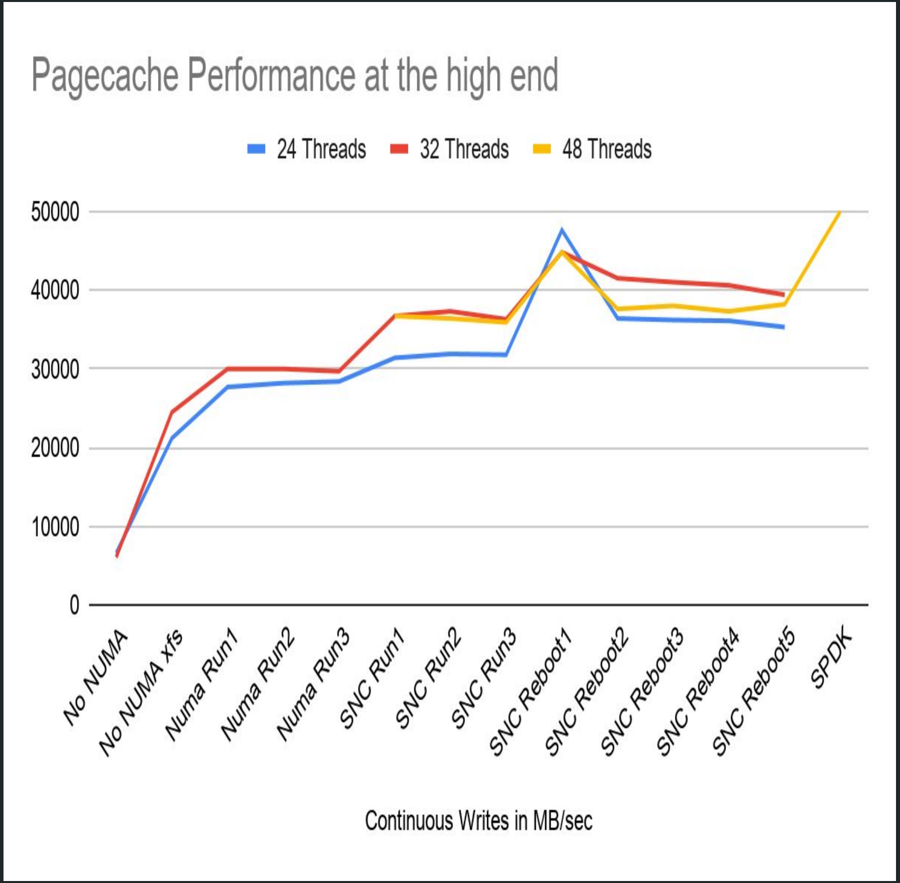
XFS was developed for performance and is required for high performance setups



Top Performance

The larger the parallelism in the system through additional NUMA Nodes the higher the speed

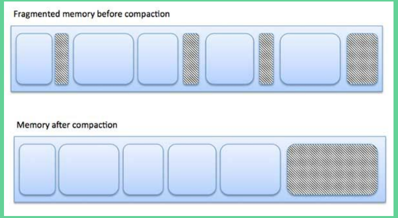
Top Performance requires contiguous memory which is only available after a reboot



Dealing with Memory Fragmentation

- *Reboot*
  Most contiguous memory is available after reboot. After that performance is gradually declining.
- *Drop Page Cache* and Metadata caches.
  - **echo 3 >/proc/sys/vm/drop_caches**

  Frees as much memory as possible in the <u>hope</u> that Linux can generate large contiguous memory. Requires re-reading data from storage.



Aha!
Close to Hardware Performance is possible even with the Page Cache

How?

1. Localize data paths to storage
2. Avoid cross segment accesses.
3. Reboot for ultimate performance
4. Or do not use systems with tiny base page sizes (Sorry, yes it is difficult to get around using Intel systems)

In order to operate effectively with large data sets the Operating System needs to operate on *large contiguous segments* of memory

With some restrictions we can scale up the page cache. However, this means adapting the application to be able to use parallel data paths to storage. An alternate approach is Direct I/O. That in turn requires other distinct modifications to the applications.
The best solution would be to have the OS do what is necessary for high performance. We need:

- Contiguous memory
- Larger chunks of memory than 4k managed by the OS
- If we want to support multiple page sizes then we need the ability to defragment memory. A fundamental change how we manage objects in the kernel. They would need to be movable in order to recover contiguous memory areas to be able to consistently be able to provide larger page sizes than the basic page.

There is work in progress in the Linux kernel by a few developers. However, significant work is required. At this point the best guestimate is that it will take at least another 2 years to implement these things upstream. This means another 5 years are required until the Linux distributions will have these features so this is the time for workarounds and improvisation.

What next?

- Be aware of the limitations of the Numaifying the page cache
- Application changes required.
- Help with I/O Subsystem changes please.
- Larger Page size please.
- Generally we need more developers to work on managing objects in larger sizes in the kernel.