Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

Conference Paper · July 2013

DOI: 10.1145/2485732.2485740

4 authors, including:

Matias Bjørling
IT University of Copenhagen
13 PUBLICATIONS **261** CITATIONS
SEE PROFILE

Philippe Bonnet
IT University of Copenhagen
104 PUBLICATIONS **2,955** CITATIONS
SEE PROFILE

Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

Matias Bjørling[*†] Jens Axboe[†] David Nellans[†] Philippe Bonnet[*]

[*]IT University of Copenhagen
{mabj,phbo}@itu.dk

[†]Fusion-io
{jaxboe,dnellans}@fusionio.com

ABSTRACT

The IO performance of storage devices has accelerated from hundreds of IOPS five years ago, to hundreds of thousands of IOPS today, and tens of millions of IOPS projected in five years. This sharp evolution is primarily due to the introduction of NAND-flash devices and their data parallel design. In this work, we demonstrate that the block layer within the operating system, originally designed to handle thousands of IOPS, has become a bottleneck to overall storage system performance, specially on the high NUMA-factor processors systems that are becoming commonplace. We describe the design of a next generation block layer that is capable of handling tens of millions of IOPS on a multi-core system equipped with a single storage device. Our experiments show that our design scales graciously with the number of cores, even on NUMA systems with multiple sockets.

Categories and Subject Descriptors

D.4.2 [**Operating System**]: Storage Management—*Secondary storage*; D.4.8 [**Operating System**]: Performance—*measurements*

General Terms

Design, Experimentation, Measurement, Performance.

Keywords

Linux, Block Layer, Solid State Drives, Non-volatile Memory, Latency, Throughput.

1 Introduction

As long as secondary storage has been synonymous with hard disk drives (HDD), IO latency and throughput have been shaped by the physical characteristics of rotational devices: Random accesses that require disk head movement are slow and sequential accesses that only require rotation of the disk platter are fast. Generations of IO intensive algorithms and systems have been designed based on these two fundamental characteristics. Today, the advent of solid state disks (SSD) based on non-volatile memories (NVM) (e.g., flash or phase-change memory [11, 6]) is transforming the performance characteristics of secondary storage. SSDs often exhibit little latency difference between sequential and random IOs [16]. IO latency for SSDs is in the order of tens of microseconds as opposed to tens of milliseconds for HDDs. Large internal data parallelism in SSDs disks enables many concurrent IO operations which, in turn, allows single devices to achieve close to a million IOs per second (IOPS) for random accesses, as opposed to just hundreds on traditional magnetic hard drives. In Figure 1, we illustrate the evolution of SSD performance over the last couple of years.

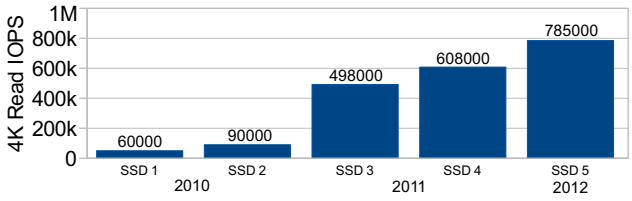

Figure 1: IOPS for 4K random read for five SSD devices.

A similar, albeit slower, performance transformation has already been witnessed for network systems. Ethernet speed evolved steadily from 10 Mb/s in the early 1990s to 100 Gb/s in 2010. Such a regular evolution over a 20 years period has allowed for a smooth transition between lab prototypes and mainstream deployments over time. For storage, the rate of change is much faster. We have seen a 10,000x improvement over just a few years. The throughput of modern storage devices is now often limited by their hardware (i.e., SATA/SAS or PCI-E) and software interfaces [28, 26]. Such rapid leaps in hardware performance have exposed previously unnoticed bottlenecks at the software level, both in the operating system and application layers. Today, with Linux, a single CPU core can sustain an IO submission rate of around 800 thousand IOPS. Regardless of how many cores are used to submit IOs, the operating system block layer can not scale up to over one million IOPS. This may be fast enough for today's SSDs - but not for tomorrow's.

We can expect that **(a)** SSDs are going to get faster, by increasing their internal parallelism[1] [9, 8] and **(b)** CPU

SYSTOR '13 June 30 - July 02 2013, Haifa, Israel

[1]If we look at the performance of NAND-flash chips, access times are getting slower, not faster, in timings [17]. Access time, for individual flash chips, increases with shrinking feature size, and increasing number of dies per package. The decrease in individual chip performance is compensated by improved parallelism within and across chips.

performance will improve largely due to the addition of more cores, whose performance may largely remain stable [24, 27].

If we consider a SSD that can provide 2 million IOPS, applications will no longer be able to fully utilize a single storage device, regardless of the number of threads and CPUs it is parallelized across due to current limitations within the operating system.

Because of the performance bottleneck that exists today within the operating system, some applications and device drivers are already choosing to bypass the Linux block layer in order to improve performance [8]. This choice increases complexity in both driver and hardware implementations. More specifically, it increases duplicate code across error-prone driver implementations, and removes generic features such as IO scheduling and quality of service traffic shaping that are provided by a common OS storage layer.

Rather than discarding the block layer to keep up with improving storage performance, we propose a new design that fixes the scaling issues of the existing block layer, while preserving its best features. More specifically, our contributions are the following:

1. We recognize that the Linux block layer has become a bottleneck (we detail our analysis in Section 2). The current design employs a single coarse lock design for protecting the request queue, which becomes the main bottleneck to overall storage performance as device performance approaches 800 thousand IOPS. This single lock design is especially painful on parallel CPUs, as all cores must agree on the state of the request queue lock, which quickly results in significant performance degradation.

2. We propose a new design for IO management within the block layer. Our design relies on multiple IO submission/completion queues to minimize cache coherence across CPU cores. The main idea of our design is to introduce two levels of queues within the block layer: (i) software queues that manage the IOs submitted from a given CPU core (e.g., the block layer running on a CPU with 8 cores will be equipped with 8 software queues), and (ii) hardware queues mapped on the underlying SSD driver submission queue.

3. We evaluate our multi-queue design based on a functional implementation within the Linux kernel. We implement a new no-op block driver that allows developers to investigate OS block layer improvements. We then compare our new block layer to the existing one on top of the noop driver (thus focusing purely on the block layer performance). We show that a two-level locking design reduces the number of cache and pipeline flushes compared to a single level design, scales gracefully in high NUMA-factor architectures, and can scale up to 10 million IOPS to meet the demand of future storage products.

The rest of the paper is organized as follows: In Section 2 we review the current implementation of the Linux block layer and its performance limitations. In Section 3 we propose a new multi-queue design for the Linux block layer. In Section 4 we describe our experimental framework, and in Section 5, we discuss the performance impact of our multi-queue design. We discuss related work in Section 6, before drawing our conclusions in Section 7.

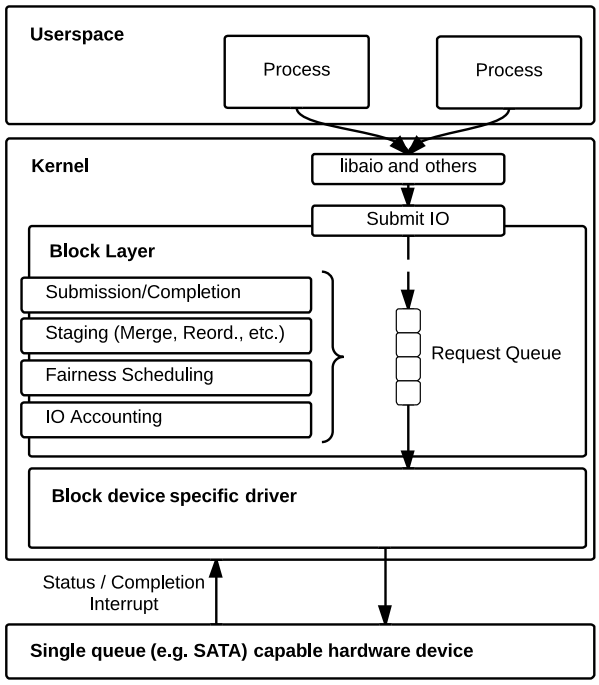

Figure 2: Current single queue Linux block layer design.

2 OS Block Layer

Simply put, the operating system block layer is responsible for shepherding IO requests from applications to storage devices [2]. The block layer is a glue that, on the one hand, allows applications to access diverse storage devices in a uniform way, and on the other hand, provides storage devices and drivers with a single point of entry from all applications. It is a convenience library to hide the complexity and diversity of storage devices from the application while providing common services that are valuable to applications. In addition, the block layer implements IO-fairness, IO-error handling, IO-statistics, and IO-scheduling that improve performance and help protect end-users from poor or malicious implementations of other applications or device drivers.

2.1 Architecture

Figure 2 illustrates the architecture of the current Linux block layer. Applications submit IOs via a kernel system call, that converts them into a data structure called a block IO. Each block IO contains information such as IO address, IO size, IO modality (read or write) or IO type (synchronous/ asynchronous)[2]. It is then transferred to either libaio for asynchronous IOs or directly to the block layer for synchronous IO that submit it to the block layer. Once an IO request is submitted, the corresponding block IO is buffered in the staging area, which is implemented as a queue, denoted the *request queue*.

Once a request is in the staging area, the block layer may perform IO scheduling and adjust accounting information before scheduling IO submissions to the appropriate storage

[2]See *include/linux/blk_types.h* in the Linux kernel (kernel.org) for a complete description of the Block IO data structure.

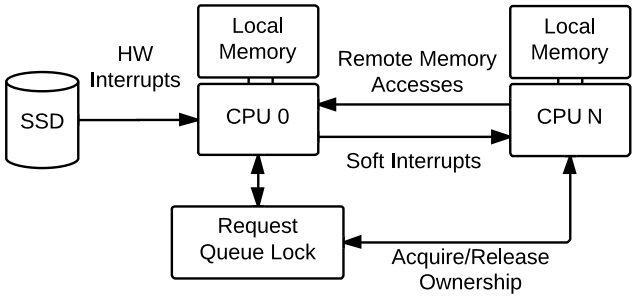

Figure 3: Simplified overview of bottlenecks in the block layer on a system equipped with two cores and a SSD.

device driver. Note that the Linux block layer supports pluggable IO schedulers: noop (no scheduling), deadline-based scheduling [12], and CFQ [10] that can all operate on IO within this staging area. The block layer also provides a mechanism for dealing with IO completions: each time an IO completes within the device driver, this driver calls up the stack to the generic completion function in the block layer. In turn the block layer then calls up to an IO completion function in the libaio library, or returns from the synchronous read or write system call, which provides the IO completion signal to the application.

With the current block layer, the staging area is represented by a request queue structure. One such queue is instantiated per block device. Access is uniform across all block devices and an application need not know what the control flow pattern is within the block layer. A consequence of this single queue per device design however is that the block layer cannot support IO scheduling across devices.

2.2 Scalability

We analyzed the Linux kernel to evaluate the performance of the current block layer on high performance computing systems equipped with high-factor NUMA multi-core processors and high IOPS NAND-flash SSDs. We found that the block layer had a considerable overhead for each IO; Specifically, we identified three main problems, illustrated in Figure 3:

1. *Request Queue Locking*: The block layer fundamentally synchronizes shared accesses to an exclusive resource: the IO request queue. (i) Whenever a block IO is inserted or removed from the request queue, this lock must be acquired. (ii) Whenever the request queue is manipulated via IO submission, this lock must be acquired. (iii) As IOs are submitted, the block layer proceeds to optimizations such as plugging (letting IOs accumulate before issuing them to hardware to improve cache efficiency), (iv) IO reordering, and (v) fairness scheduling. Before any of these operations can proceed, the request queue lock must be acquired. This is a major source of contention.

2. *Hardware Interrupts*: The high number of IOPS causes a proportionally high number of interrupts. Most of today's storage devices are designed such that one core (within CPU 0 on Figure 3) is responsible for handling all hardware interrupts and forwarding them to other cores as soft interrupts regardless of the CPU issuing and completing the IO. As a result, a single core may spend considerable time in handling these interrupts, context switching, and polluting L1 and L2 caches that applications could rely on for data locality [31]. The other cores (within CPU N on Figure 3) then also must take an IPI to perform the IO completion routine. As a result, in many cases two interrupts and context switches are required to complete just a single IO.

3. *Remote Memory Accesses*: Request queue lock contention is exacerbated when it forces remote memory accesses across CPU cores (or across sockets in a NUMA architecture). Such remote memory accesses are needed whenever an IO completes on a different core from the one on which it was issued. In such cases, acquiring a lock on the request queue to remove the block IO from the request queue incurs a remote memory access to the lock state stored in the cache of the core where that lock was last acquired, the cache line is then marked shared on both cores. When updated, the copy is explicitly invalidated from the remote cache. If more than one core is actively issuing IO and thus competing for this lock, then the cache line associated with this lock is continuously bounced between those cores.

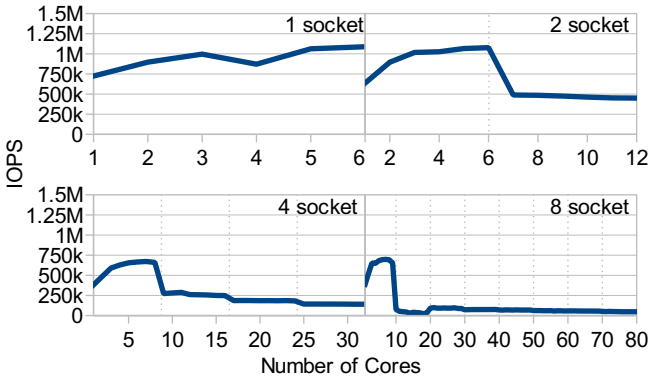

Figure 4: IOPS throughput of Linux block layer as a function of number of CPU's issuing IO. Divided into 1, 2, 4 and 8 socket systems. Note: Dotted line show socket divisions.

Figure 4 shows 512 bytes IOs being submitted to the kernel as fast as possible; IOPS throughput is depicted as a function of the number of CPU's that are submitting and completing IOs to a single device simultaneously. We observe that when the number of processes is lower than the number cores on a single socket (i.e., 6), throughput improves, or is at least maintained, as multiple CPU's issue IOs. For 2, 4, and 8-socket architectures which have largely supplanted single socket machines in the HPC space, when IOs are issued from a CPU that is located on a remote socket (and typically NUMA node), absolute performance drops substantially regardless the absolute number of sockets in the system.

Remote cacheline invalidation of the request queue lock is significantly more costly on complex four and eight socket systems where the NUMA-factor is high and large cache directory structures are expensive to access. On four and eight

socket architectures, the request queue lock contention is so high that multiple sockets issuing IOs reduces the throughput of the Linux block layer to just about 125 thousand IOPS even though there have been high end solid state devices on the market for several years able to achieve higher IOPS than this. The scalability of the Linux block layer is not an issue that we might encounter in the future, it is a significant problem being faced by HPC in practice today.

3 Multi-Queue Block Layer

As we have seen in Section 2.2, reducing lock contention and remote memory accesses are key challenges when redesigning the block layer to scale on high NUMA-factor architectures. Dealing efficiently with the high number of hardware interrupts is beyond the control of the block layer (more on this below) as the block layer cannot dictate how a device driver interacts with its hardware. In this Section, we propose a two-level multi-queue design for the Linux block layer and discuss its key differences and advantages over the current single queue block layer implementation. Before we detail our design, we summarize the general block layer requirements.

3.1 Requirements

Based on our analysis of the Linux block layer, we identify three major requirements for a block layer:

- **Single Device Fairness**

  Many application processes may use the same device. It is important to enforce that a single process should not be able to starve all others. This is a task for the block layer. Traditionally, techniques such as CFQ or deadline scheduling have been used to enforce fairness in the block layer. Without a centralized arbiter of device access, applications must either coordinate among themselves for fairness or rely on the fairness policies implemented in device drivers (which rarely exist).

- **Single and Multiple Device Accounting**

  The block layer should make it easy for system administrators to debug or simply monitor accesses to storage devices. Having a uniform interface for system performance monitoring and accounting enables applications and other operating system components to make intelligent decisions about application scheduling, load balancing, and performance. If these were maintained directly by device drivers, it would be nearly impossible to enforce the convenience of consistency application writers have become accustom to.

- **Single Device IO Staging Area**

  To improve performance and enforce fairness, the block layer must be able to perform some form of IO scheduling. To do this, the block layer requires a staging area, where IOs may be buffered before they are sent down into the device driver. Using a staging area, the block layer can reorder IOs, typically to promote sequential accesses over random ones, or it can group IOs, to submit larger IOs to the underlying device. In addition, the staging area allows the block layer to adjust its submission rate for quality of service or due to device back-pressure indicating the OS should not send down additional IO or risk overflowing the device's buffering capability.

3.2 Our Architecture

The key insight to improved scalability in our multi-queue design is to distribute the lock contention on the single request queue lock to multiple queues through the use of two levels of queues with distinct functionally as shown in Figure 5:

- *Software Staging Queues.* Rather than staging IO for dispatch in a single software queue, block IO requests are now maintained in a collection of one or more request queues. These staging queues can be configured such that there is one such queue per socket, or per core, on the system. So, on a NUMA system with 4 sockets and 6 cores per socket, the staging area may contain as few as 4 and as many as 24 queues. The variable nature of the request queues decreases the proliferation of locks if contention on a single queue is not a bottleneck. With many CPU architectures offering a large shared L3 cache per socket (typically a NUMA node as well), having just a single queue per processor socket offers a good trade-off between duplicated data structures which are cache unfriendly and lock contention.

- *Hardware Dispatch Queues.* After IO has entered the staging queues, we introduce a new intermediate queuing layer known as the hardware dispatch queues. Using these queues block IOs scheduled for dispatch are not sent directly to the device driver, they are instead sent to the hardware dispatch queue. The number of hardware dispatch queues will typically match the number of hardware contexts supported by the device driver. Device drivers may choose to support anywhere from one to 2048 queues as supported by the message signaled interrupts standard MSI-X [25]. Because IO ordering is not supported within the block layer any software queue may feed any hardware queue without needing to maintain a global ordering. This allows hardware to implement one or more queues that map onto NUMA nodes or CPU's directly and provide a fast IO path from application to hardware that never has to access remote memory on any other node.

This two level design explicitly separates the two buffering functions of the staging area that was previously merged into a single queue in the Linux block layer: (i) support for IO scheduling (software level) and (ii) means to adjust the submission rate (hardware level) to prevent device buffer over run.

The number of entries in the software level queue can dynamically grow and shrink as needed to support the outstanding queue depth maintained by the application, though queue expansion and contraction is a relatively costly operation compared to the memory overhead of maintaining enough free IO slots to support most application use. Conversely, the size of the hardware dispatch queue is bounded and correspond to the maximum queue depth that is supported by the device driver and hardware. Today many SSD's that support native command queuing support a queue depth of just 32, though high-end SSD storage devices may have much deeper queue support to make use of the high internal parallelism of their flash architecture. The 32 in-flight request limit found on many consumer SSD's is likely to increase substantially to support increased IOPS rates as

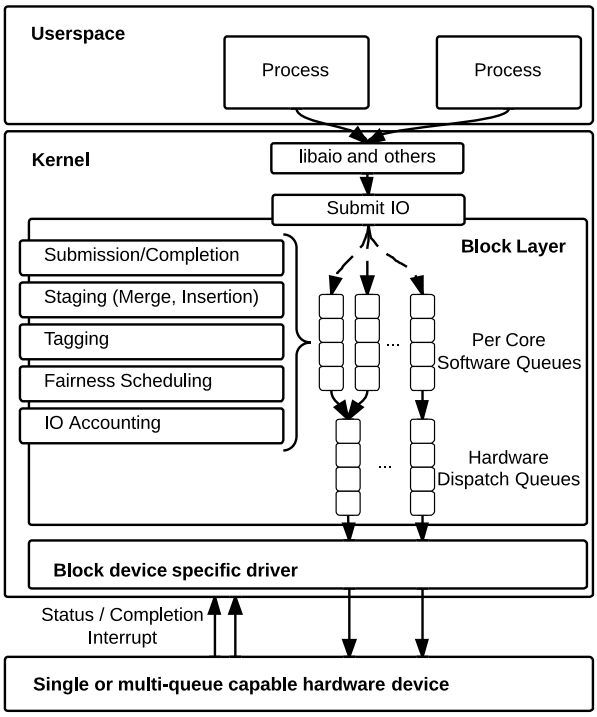

Figure 5: Proposed two level Linux block layer design.

a 1 million IOPS capable device will cause 31 thousand context switches per second simply to process IO in batches of 32. The CPU overhead of issuing IO to devices is inversely proportional to the amount of IO that is batched in each submission event.

3.2.1 IO-Scheduling

Within the software queues, IOs can be shaped by per CPU or NUMA node policies that need not access local memory. Alternatively, policies may be implemented across software queues to maintain global QoS metrics on IO, though at a performance penalty. Once the IO has entered the hardware dispatch queues, reordering i/Nums no longer possible. We eliminate this possibility so that the only contenders for the hardware dispatch queue are inserted to the head of the queue and removed from the tail by the device driver, thus eliminating lock acquisitions for accounting or IO-scheduling. This improves the fast path cache locality when issuing IO's in bulk to the device drivers.

Our design has significant consequences on how IO may be issued to devices. Instead of inserting requests in the hardware queue in sorted order to leverage sequential accesses (which was a main issue for hard drives), we simply follow a FIFO policy: we insert the incoming block IO submitted by core i at the top of the request queue attached to core i or the NUMA socket this core resides on. Traditional IO-schedulers have worked hard to turn random into sequential access to optimize performance on traditional hard drives. Our two level queuing strategy relies on the fact that modern SSD's have random read and write latency that is as fast as their sequential access. Thus interleaving IOs from multiple software dispatch queues into a single hardware dispatch queue does not hurt device performance. Also, by inserting requests into the local software request queue, our design respects thread locality for IOs and their completion.

While global sequential re-ordering is still possible across the multiple software queues, it is only necessary for HDD based devices, where the additional latency and locking overhead required to achieve total ordering does not hurt IOPS performance. It can be argued that, for many users, it is no longer necessary to employ advanced fairness scheduling as the speed of the devices are often exceeding the ability of even multiple applications to saturate their performance. If fairness is essential, it is possible to design a scheduler that exploits the characteristics of SSDs at coarser granularity to achieve lower performance overhead [23, 13, 19]. Whether the scheduler should reside in the block layer or on the SSD controller is an open issue. If the SSD is responsible for fair IO scheduling, it can leverage internal device parallelism, and lower latency, at the cost of additional interface complexity between disk and OS [8, 4].

3.2.2 Number of Hardware Queues

Today, most SATA, SAS and PCI-E based SSDs, support just a single hardware dispatch queue and a single completion queue using a single interrupt to signal completions. One exception is the upcoming NVM Express (NVMe) [18] interface which supports a flexible number of submission queues and completion queues. For devices to scale IOPS performance up, a single dispatch queue will result in cross CPU locking on the dispatch queue lock, much like the previous request queue lock. Providing multiple dispatch queues such that there is a local queue per NUMA node or CPU will allow NUMA local IO path between applications and hardware, decreasing the need for remote memory access across all subsections of the block layer. In our design we have moved IO-scheduling functionality into the software queues only, thus even legacy devices that implement just a single dispatch queue see improved scaling from the new multi-queue block layer.

3.2.3 Tagged IO and IO Accounting

In addition to introducing a two-level queue based model, our design incoporates several other implementation improvements. First, we introduce tag-based completions within the block layer. Device command tagging was first introduced with hardware supporting native command queuing. A tag is an integer value that uniquely identifies the position of the block IO in the driver submission queue, so when completed the tag is passed back from the device indicating which IO has been completed. This eliminates the need to perform a linear search of the in-flight window to determine which IO has completed.

In our design, we build upon this tagging notion by allowing the block layer to generate a unique tag associated with an IO that is inserted into the hardware dispatch queue (between size 0 and the max dispatch queue size). This tag is then re-used by the device driver (rather than generating a new one, as with NCQ). Upon completion this same tag can then be used by both the device driver and the block layer to identify completions without the need for redundant tagging. While the MQ implementation could maintain a traditional in-flight list for legacy drivers, high IOPS drivers will likely need to make use of tagged IO to scale well.

Second, to support fine grained IO accounting we have modified the internal Linux accounting library to provide statistics for the states of both the software queues and dis-

patch queues. We have also modified the existing tracing and profiling mechanisms in blktrace, to support IO tracing for future devices that are multi-queue aware. This will allow future device manufacturers to optimize their implementations and provide uniform global statistics to HPC customers whose application performance is increasingly dominated by the performance of the IO-subsystem.

3.3 Multiqueue Impact on Device Manufacturers

One drawback of the our design is that it will require some extensions to the bottom edge device driver interface to achieve optimal performance. While the basic mechanisms for driver registration and IO submission/completion remain unchanged, our design introduces these following requirements:

- *HW dispatch queue registration*: The device driver must export the number of submission queues that it supports as well as the size of these queues, so that the block layer can allocate the matching hardware dispatch queues.
- *HW submission queue mapping function*: The device driver must export a function that returns a mapping between a given software level queue (associated to core i or NUMA node i), and the appropriate hardware dispatch queue.
- *IO tag handling*: The device driver tag management mechanism must be revised so that it accepts tags generated by the block layer. While not strictly required, using a single data tag will result in optimal CPU usage between the device driver and block layer.

These changes are minimal and can be implemented in the software driver, typically requiring no changes to existing hardware or software. While optimal performance will come from maintaining multiple hardware submission and completion queues, legacy devices with only a single queue can continue to operate under our new Linux block layer implementation.

4 Experimental Methodology

In the remaining of the paper, we denote the existing block layer as single queue design (SQ), our design as the multi-queue design (MQ), and a driver which bypasses the Linux block layer as Raw. We implemented the MQ block layer as a patch to the Linux kernel 3.10[3].

4.1 Hardware Platforms

To conduct these comparisons, we rely on a *null* device driver, i.e., a driver that is not connected to an underlying storage device. This null driver simply receives IOs as fast as possible and acknowledges completion immediately. This pseudo block device can acknowledge IO requests faster than even a DRAM backed physical device, making the null block device an ideal candidate for establishing an optimal baseline for scalability and implementation efficiency.

Using the null block device, we experiment with 1, 2, 4 and 8 sockets systems, i.e., Sandy Bridge-E, Westmere-EP, Nehalem-EX and Westmere-EX Intel platforms. Table 1 summarizes the characteristics of these four platforms. The 1, 2 and 4-sockets systems use direct QPI links as interconnect between sockets, while the 8-nodes system has a lower and upper CPU board (with 4 sockets each) and an interconnect board for communication. We disabled the turbo boost and hyper-threading CPU features as well as any ACPI C and P-state throttling on our systems to decrease the variance in our measurements that would be caused by power savings features.

Platform/Intel	Sandy Bridge-E	Westmere-EP	Nehalem-EX	Westmere-EX
Processor	i7-3930K	X5690	X7560	E7-2870
Num. of Cores	6	12	32	80
Speed (Ghz)	3.2	3.46	2.66	2.4
L3 Cache (MB)	12	12	24	30
NUMA nodes	1	2	4	8

Table 1: Architecture of Evaluation Systems

4.2 IO Load Generation

We focus our evaluations on latency and throughput. We experiment with latency by issuing a single IO per participating core at a time using the pread/pwrite interface of the Linux kernel. We experiment with throughput by overlapping the submission of asynchronous IOs. In the throughput experiment we sustain 32 outstanding IOs per participating core, i.e., if 8 cores are issuing IOs, then we maintain 256 outstanding IOs. We use 32 IOs per process context because it matches the requirements of today's SSD devices. Our IO-load is generated using the flexible io generator (fio) [14] that allows us to carefully control the queue-depth, type, and distribution of IO onto a the LBA space of the block device. In all experiments we use 512 bytes read IO's, though the type of IO is largely irrelevant since the null block driver does not perform any computation or data transfer, it simply acknowledges all requests immediately.

4.3 Performance Metrics

The primary metrics for our experiments are absolute throughput (IOPS) and latency (μ-seconds) of the block layer.

5 Results

In a first phase, we compare our new block layer design (MQ) with the existing Linux block layer (SQ), and the optimal baseline (Raw). In a second phase, we investigate how our design allows the block layer to scale as the number of available cores in the system increases. We leave a performance tuning study of MQ (e.g., quality of the performance optimizations within the block layer) as a topic for future work.

For each system configuration, we create as many fio processes as there are cores and we ensure that all cores are utilized 100%. For the 1 socket system, the maximum number of cores is 6. For the 2 (resp., 4 and 8) sockets system, the maximum number of core is 12 (resp., 32 and 80), and we mark the separation between both 6 (resp., 8 and 10) cores sockets with a vertical dotted line. Unless otherwise noted, for MQ, a software queue is associated to each core and a hardware dispatch queue is associated to each socket.

[3]Our implementation is available online at `http://git.kernel.dk/?p=linux-block.git;a=shortlog;h=refs/heads/new-queue`

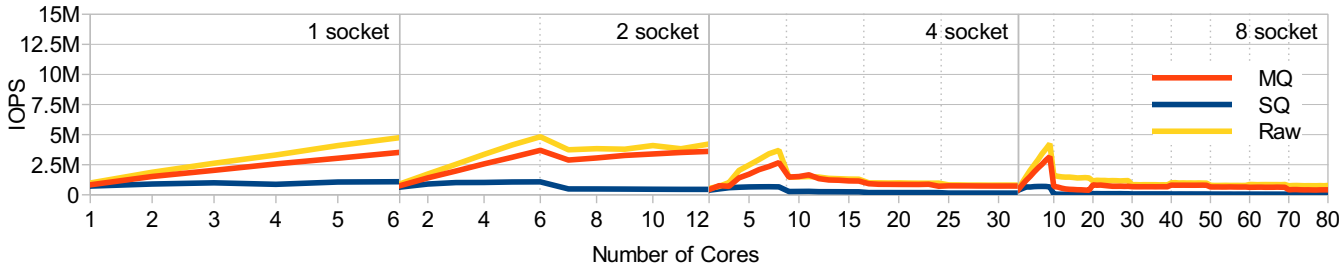

Figure 6: IOPS for single/multi-queue and raw on the 1, 2, 4 and 8-nodes systems.

5.1 Comparing MQ, SQ and Raw

Figure 6 presents throughput (in IOPS) for SQ, MQ and Raw as a function of the number of cores available in the 1 socket, 2-sockets, 4-sockets, and 8-sockets systems respectively. Overall, we can make the following observations. First, with the single queue block layer implementation, throughput is limited below 1 million IOPS regardless of the number of CPUs issuing IO or of the number of sockets in the system. *The current Linux block layer implementation can not sustain more than 1 million IOPS on a single block device.*

Second, our new two layer multi-queue implementation improves performance significantly. The system can sustain up to 3.5 million IOPS on a single socket system. However, in multi-socket systems scaling does not continue at nearly the same rate.

Let us analyze those results in more details:

1. The scalability problems of SQ are evident as soon as more than one core is used in the system. Additional cores spend most of their cycles acquiring and releasing spin locks for the single request queue and as such do not contribute to improving throughput. This problem gets even worse on multi-socket systems, because their inter-connects and the need to maintain cache-coherence.

2. MQ performance is similar to SQ performance on a single core. This shows that the overhead of introducing multiple queues is minimal.

3. MQ scales linearly within one socket. This is because we removed the need for the block layer to rely on synchronization between cores when block IOs are manipulated inside the software level queues.

4. For all systems, MQ follows the performance of Raw closely. MQ is in fact a constant factor away from the raw measurements, respectively 22%, 19%, 13% and 32% for the 1, 2, 4, 8-sockets systems. This overhead might seem large, but the raw baseline does not implement logic that is required in a real device driver. For good scalability, the MQ performance just needs to follow the trend of the baseline.

5. The scalability of MQ and raw exhibits a sharp dip when the number of sockets is higher than 1. We see that throughput reaches 5 million IOPS (resp., 3.8 and 4) for 6 cores (resp., 7 and 9) on a 2 sockets system (resp., 4 and 8 sockets system). This is far from the 10 million IOPS that we could have hoped for. Interestingly, MQ follows roughly the raw baseline. *There is thus a problem of scalability, whose root lies outside the block layer, that has a significant impact on performance. We focus on this problem in the next Section.*

	1 socket	2 sockets	4 sockets	8 sockets
SQ	50 ms	50 ms	250 ms	750 ms
MQ	50 ms	50 ms	50 ms	250 ms
Raw	50 ms	50 ms	50 ms	250 ms

Table 2: Maximum latency for each of the systems.

Let us now turn our attention to latency. As we explained in the previous section, latency is measured through synchronous IOs (with a single outstanding IO per participating core). The latency is measured as the time it takes to go from the application, through the kernel system call, into the block layer and driver and back again. Figure 7 shows average latency (in μ-seconds) as a function of the number of cores available for the four systems that we study.

Ideally, latency remains low regardless of the number of cores. In fact, remote memory accesses contribute to increase latency on multi-sockets systems. For SQ, we observe that latency increases linearly with the number of cores, slowly within one socket, and sharply when more than one socket is active. For MQ, latency remains an order of magnitude lower than for SQ. This is because, for MQ, the only remote memory accesses that are needed are those concerning the hardware dispatch queue (there is no remote memory accesses for synchronizing the software level queues). Note that, on 8 sockets system, the SQ graph illustrates the performance penalty which is incurred when crossing the interconnect board (whenever 2, 4, 6 and 8 sockets are involved).

Table 2 shows the maximum latency across all experiments. With SQ, the maximum latency reaches 250 milliseconds in the 4 sockets system and 750 milliseconds on the 8 sockets system. Interestingly, with SQ on a 8 sockets systems, 20% of the IO requests take more than 1 millisecond to complete. This is a very significant source of variability for IO performance. In contrast, with MQ, the number of IOs which take more than 1ms to complete only reaches 0.15% for an 8 socket system, while it is below 0.01% for the other systems. Note Raw exhibits minimal, but stable, variation across all systems with around 0.02% of the IOs that take more than 1ms to complete.

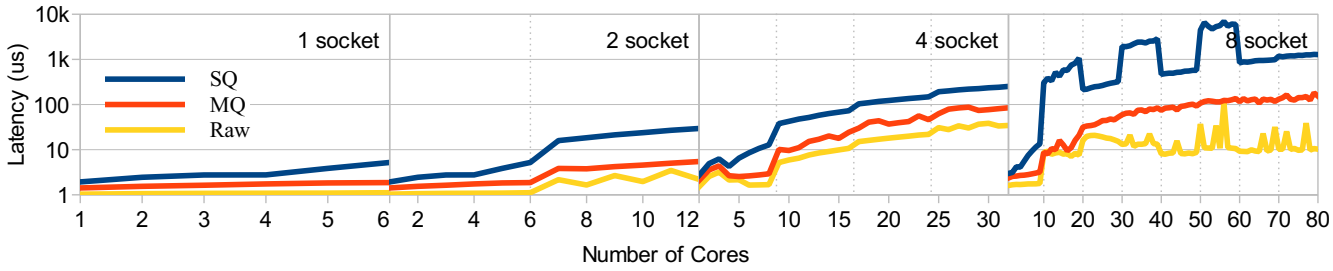

Figure 7: Latency on the 1, 2, 4 and 8 node system using the null device.

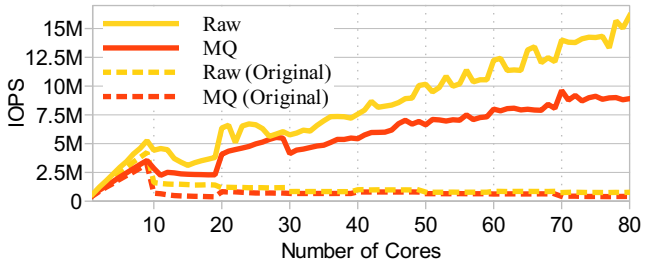

Figure 8: IOPS for MQ and raw with libaio fixes applied on the 8-nodes systems.

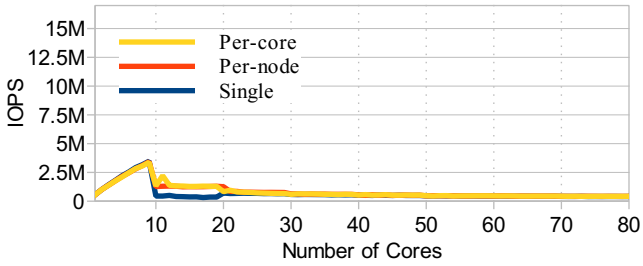

Figure 9: IOPS for a single software queue with varied number of mapped hardware dispatch queues on the 8 socket system.

5.2 Improving Application Level IO Submission

The throughput graphs from the previous Section exposed scalability problem within the Linux stack, on top of the block layer. Through profiling we were able to determine that the asynchronous (libaio) and direct IO layers, used within the kernel to transfer block IOs from userspace into to the block layer, have several bottlenecks that have are first being exposed with the new MQ block layer implementation. These bottlenecks are: (i) a context list lock is issued for each request, (ii) a completion ring in libaio used to manage sleep/wakeup cycles and (iii) a number of shared variables are being updated throughout the library. We removed these bottlenecks through a series of implementation improvements.

First, we replaced the context list lock with a lockless list, which instead of using mutexes to update a variable used the compare-and-swap instruction of the processor to perform the update. Second, we eliminated the use of the completion ring as it caused an extra lock access when updating the number of elements in the completion list, which in the worst case, could put the application process to sleep. Third, we used atomic compare-and-swap instructions to manipulate the shared counters for internal data structures (e.g. the number of users of AIO context) instead of the native mutex structures.

Figure 8 demonstrates the IOPS of the raw and MQ designs using this new userspace IO submission library, on the 8-socket system, which has the hardest time maintaining IO scalability. We observe that both the MQ and Raw implementations, while still losing efficiency when moving to a second socket, are able to scale IOPS near linearly up to the maximum number of cores within the system. The multi-queue design proposed here allows the block layer to scale up to 10 million IOPS utilizing 70s cores on an 8 socket NUMA system while maintaining the conveniences of the block layer implementation for application compatibility. We recognize that the efficiency of the MQ (and Raw) implementations drops significantly when moving from one socket onto a second. This indicates that there are further bottlenecks to be improved upon in Linux that lay outside the block layer. Possible candidates are interrupt handling, context switching improvements, and other core OS functions that we leave for future work.

5.3 Comparing Allocation of Software and Hardware Dispatch Queues

As our design introduces two levels of queues (the software and hardware dispatch queues), we must investigate how number of queues defined for each level impacts performance. We proceed in two steps. First, we fix the number of software level queues to one and we vary the number of hardware dispatch queues. Second, we fix the number of software level queues to one per core and we vary the number of hardware dispatch queues. In both experiments, the number of hardware dispatch queues is either one, one per core (denoted per-core) or one per socket (denoted per-socket). All experiments with the MQ block layer on on the 8-socket system.

Figure 9 presents throughput using a single software queue. We observe that all configurations show a sharp performance dip when the second socket is introduced. Furthermore, we observe that a single software and hardware dispatch queue perform significantly worse on the second socket, but follow each other when entering the third socket. Overall, this experiment shows that a single software queue does not allow the block layer to scape gracefully.

We show in Figure 10 the results of our experiments with

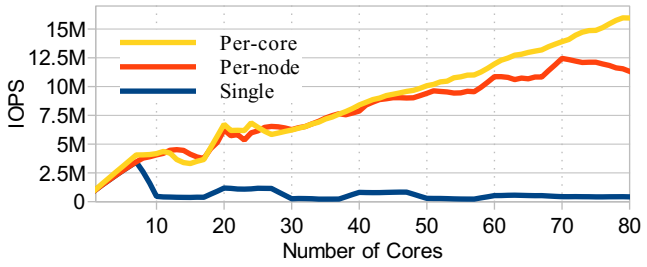

Figure 10: IOPS for per-core software queue with a different number of hardware dispatch queues.

a software queue per core. We see that combining multiple software and hardware dispatch queues enable high performance, reaching more than 15 million IOPS using the least contended per-core/per-core queue mapping. The per-node hardware dispatch queue configuration also scales well up to the fifth socket, but then the per-node slowly decrease in throughput. This occur when the socket interconnect becomes the bottleneck. Further performance is possible as more processes are added, but they slightly suffer from less available bandwidth. To achieve the highest throughput, per-core queues for both software and hardware dispatch queues are advised. This is easily implemented on the software queue side, while hardware queues must be implemented by the device itself. Hardware vendors can restrict the number of hardware queues to the system sockets available and still provide scalable performance.

6 Related Work

Our redesign of the block layer touch on network, hardware interfaces and NUMA systems. Below we describe related work in each of these fields.

6.1 Network

The scalability of operating system network stacks has been addressed in the last 10 years by incorporating multiple sender and receiver queues within a single card to allow a proliferation of network ports [1, 21]. This allows a single driver to manage multiple hardware devices and reduce code and data structure duplication for common functionality. Our work builds upon the foundation of networking multi-queue designs by allowing a single interface point, the block device, with multiple queues within the software stack.

Optimization of the kernel stack itself, with the purpose of removing IO bottlenecks, has been studied using the Moneta platform [7]. Caulfield et al. propose to bypass the block layer and implement their own driver and single queue mechanism to increase performance. By bypassing the block layer, each thread issues an IO and deals with its completion. Our approach is different, as we propose to redesign the block layer thus improving performance across all devices, for all applications.

6.2 Hardware Interface

The NVMe interface [18] attempts to address many of the scalability problems within the block layer implementation. NVMe however proposes a new dynamic interface to accommodate the increased parallelism in NVM storage on which each process has its own submission and completion queue to a NVM storage device. While this is excellent for scalability, it requires application modification and pushes much of the complexity of maintaining storage synchronization out of the operating system into the application. This also exposes security risks to the application such as denial of service without a central trusted arbitrar of device access.

6.3 NUMA

The affect of NUMA designs on parallel applications has been studied heavily in the HPC space [22, 30, 15, 20] and big data communities [29, 3, 5]. We find that many of these observations, disruptive interrupts, cache locality, and lock-contention have the same negative performance penalty within the operating system and block layer. Unfortunately, as a common implementation for all applications, some techniques to avoid lock contention such as message passing simply are in-feasible to retrofit into a production operating system built around shared memory semantics.

One approach to improving IO performance is to access devices via memory-mapped IO. While this does save some system call overhead, this does not fundamentally change or improve the scalability of the operating system block layer. Additionally, it introduces a non-powercut safe fault domain (the DRAM page-cache) that applications may be un-aware of while simultaneously requiring a large application re-write to take leverage.

7 Conclusions and Future Work

In this paper, we have established that the current design of the Linux block layer does not scale beyond one million IOPS per device. This is sufficient for today's SSD, but not for tomorrow's. We proposed a new design for the Linux block layer. This design is based on two levels of queues in order to reduce contention and promote thread locality. Our experiments have shown the superiority of our design and its scalability on multi-socket systems. Our multiqueue design leverages the new capabilities of NVM-Express or high-end PCI-E devices, while still providing the common interface and convenience features of the block layer.

We exposed limitations of the Linux IO stack beyond the block layer. Locating and removing those additional bottlenecks is a topic for future work. Future work also includes performance tuning with multiple hardware queues, and experiments with multiqueue capable hardware prototypes. As one bottleneck is removed, a new choke point is quickly created, creating an application through device NUMA-local IO-stack is an on-going process. We intend to work with device manufacturers and standards bodies to ratify the inclusion of hardware capabilities that will encourage adoption of the multiqueue interface and finalize this new block layer implementation for possible inclusion in the mainline Linux kernel.

8 References

[1] Improving network performance in multi-core systems. *Intel Corporation*, 2007.

[2] J. Axboe. Linux Block IO present and future. *Ottawa Linux Symposium*, 2004.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and S. Akhilesh. The multikernel: a new OS architecture for scalable multicore systems. *Symposium on Operating Systems Principles*, 2009.

[4] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan. The necessary death of the block device interface. In *Conference on Innovative Data Systems Research*, 2013.

[5] S. Boyd-wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. *Operating Systems Design and Implementation*, 2010.

[6] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, C. Lam, and A. Luis. Phase change memory technology. *Journal of Vacuum Science and Technology B*, 28(2):223–262, 2010.

[7] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[8] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. *SIGARCH Comput. Archit. News*, 40(1):387–400, Mar. 2012.

[9] S. Cho, C. Park, H. Oh, S. Kim, Y. Y. Yi, and G. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. Technical Report CMU-PDL-11-115, 2011.

[10] Completely Fair Queueing (CFQ) Scheduler. `http://en.wikipedia.org/wiki/CFQ`.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. *Symposium on Operating Systems Principles*, page 133, 2009.

[12] Deadline IO Scheduler. `http://en.wikipedia.org/wiki/Deadline_scheduler`.

[13] M. Dunn and A. L. N. Reddy. A new I/O scheduler for solid state devices. *Texas A&M University*, 2010.

[14] fio. `http://freecode.com/projects/fio`.

[15] P. Foglia, C. A. Prete, M. Solinas, and F. Panicucci. Investigating design tradeoffs in S-NUCA based CMP systems. *UCAS*, 2009.

[16] Fusion-io ioDrive2. `http://www.fusionio.com/`.

[17] L. M. Grupp, J. D. David, and S. Swanson. The Bleak Future of NAND Flash Memory. *USENIX Conference on File and Storage Technologies*, 2012.

[18] A. Huffman. NVM Express, Revision 1.0c. *Intel Corporation*, 2012.

[19] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk Schedulers for Solid State Drives. In *EMSOFTâĂŹ09: 7th ACM Conf. on Embedded Software*, pages 295–304, 2009.

[20] F. Liu, X. Jiang, and Y. Solihin. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. *High Performance Computer Architecture*, 2009.

[21] S. Mangold, S. Choi, P. May, O. Klein, G. Hiertz, and L. Stibor. 802.11e Wireless LAN for Quality of Service. *IEEE*, 2012.

[22] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A Non-Uniform-Memory-Access Programming Model For High-Performance Computers. *The Journal of Supercomputing*, 1996.

[23] S. Park and K. Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *USENIX Conference on File and Storage Technologies*, 2010.

[24] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006.

[25] PCI-SIG. PCI Express Specification Revision 3.0. Technical report, 2012.

[26] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[27] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[28] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[29] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[30] J. Weinberg. *Quantifying Locality In The Memory Access Patterns of HPC Applications*. PhD thesis, 2005.

[31] J. Yang, D. B. Minturn, and F. Hady. When Poll is Better than Interrupt. In *USENIX Conference on File and Storage Technologies*, 2012.