2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Energy-efficient I/O Thread Schedulers for NVMe SSDs on NUMA

Junjie Qian, Hong Jiang†, Witawas Srisa-an, Sharad Seth
University of Nebraska–Lincoln, †University of Texas–Arlington
Email: {jqian, witty, seth}@cse.unl.edu, †hong.jiang@uta.edu

Stan Skelton, Joseph Moore
NetApp Inc.
Email: {stan.skelton, joseph.moore}@netapp.com

***Abstract*—Non-volatile memory express (NVMe) based SSDs and the NUMA platform are widely adopted in servers to achieve faster storage speed and more powerful processing capability. As of now, very little research has been conducted to investigate the performance and energy efficiency of the state-of-the-art NUMA architecture integrated with NVMe SSDs, an emerging technology used to host parallel I/O threads. As this technology continues to be widely developed and adopted, we need to understand the runtime behaviors of such systems in order to design software runtime systems that deliver optimal performance while consuming only the necessary amount of energy.**

This paper characterizes the runtime behaviors of a Linux-based NUMA system employing multiple NVMe SSDs. Our comprehensive performance and energy-efficiency study using massive numbers of parallel I/O threads shows that the penalty due to CPU contention is much smaller than that due to remote access of NVMe SSDs. Based on this insight, we develop a dynamic "lesser evil" algorithm called ESN**, to minimize the impact of these two types of penalties.** ESN **is an energy-efficient profiling-based I/O thread scheduler for managing I/O threads accessing NVMe SSDs on NUMA systems. Our empirical evaluation shows that ESN can achieve optimal I/O throughput and latency while consuming up to 50% less energy and using fewer CPUs.**

I. Introduction

The traditional storage interfaces (e.g., SATA and SAS) do not provide sufficient bandwidth and concurrency [1] for the latest high-speed non-volatile memory (NVM) devices (e.g., STT-RAM, PCM, RRAM, Flash). To overcome this bottleneck, the non-volatile memory express (NVMe) technology has recently been introduced to provide wider bandwidths and efficient concurrent accesses to these NVM devices and Flash SSDs. This technology is expected to be used to satisfy the needs of the next-generation fast storage solutions [2], [3].

To fully take advantage of NVMe, designs of software components that manage data movements to and from storage modules via NVMe have also been improved. Prior work by Bjørling et al. [1] identifies the interrupts and locks in the traditional single queue block layer as the major barriers limiting the scalability of parallel I/Os. This finding led to the adoption of a multi-queue implementation in the block layer of Linux. The goal was to resolve lock contention among multiple I/O processes from different CPUs to different devices. Initially, NVMe also used multiple submission and completion queues to process I/Os in the block layer. To further improve performance, the submission and completion queues have been integrated into a hardware dispatch queue in the multi-queue block layer [4]. These two changes further promote the scalability of parallel I/O threads accessing NVM and NVMe SSDs.

While these modifications have been implemented, most recent work on performance analysis and optimization of NVMe and storage devices do not use them for two major reasons. First, NVMe-based systems are not generally available as research platforms. Second, most studies still rely on outdated software platforms (e.g., out-of-date Linux kernels or NVMe implementations) [5], [6], [7], [8], [9], [10], [11]. For example, a recent work by Dullor et al. [12] used PMEP emulator to emulate the bandwidth and latency of the NVM. However, PMEP emulator fails to characterize the NVM application in the software layer because PMEP does not use the actual NVMe connection and the multi-queue block layer in Linux. So, the reported results do not represent the most up-to-date usage of NVMe.

In addition, more servers increasingly rely on the NUMA (non-uniform memory access) architecture to achieve good overall memory performance. In a NUMA set up, NVMe modules are directly attached to the CPUs of a NUMA server in a way very similar to how DRAM modules are attached to the CPUs. In this configuration, there are local and remote accesses between NVMe SSDs and CPUs that would result in significantly different I/O throughputs and latencies. Often, past studies on this issue do not consider the latest platforms that integrate the NUMA architecture with NVMe SSDs and NVMe [9], [5]. As a result, the reported results from these studies also do not reflect the performances of the state-of-the-art NVMe-based systems.

Furthermore, outdated platforms used in those studies cannot be used to evaluate the current energy consumption of NVMe-based systems. Energy saving is important for large storage services because a significant portion of the total cost of ownership (TCO) is due to energy consumption. As such, investigation into the performance and energy characteristics of different system configurations (e.g., configuring NUMA nodes) can greatly improve the understanding of the architectural impacts, such as the mapping of I/O threads to different sockets, etc., as well as providing guidance on designs of runtime systems including I/O thread schedulers.

To illustrate this point, we present a hypothetical example. Because energy is consumed by both storage devices and pro-

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.24

IEEE computer society

cessors, an improved I/O thread scheduler, which recognizes energy impacts of different I/O requests and scheduling parameters, has a great potential to save energy [13], [14] consumed by these two components due to I/O requests and execution while maintaining optimal performance. Unfortunately, prior suggested solutions cannot achieve both. Solutions to improve energy efficiency tend to employ approaches that reduce the overall system performances (e.g., controlling voltage and frequency scaling for data-intensive applications [15], [16]).

To this end, our comprehensive investigation aims to answer the following research questions regarding trade offs between performance and energy consumption of parallel I/O threads in NVMe-based NUMA servers employing the state-of-the-art Linux kernel:

- Given the I/O workload, how are the I/O performance and energy efficiency impacted by the numbers of NVMe SSDs and CPUs and why?
- What is the impact of the different SSDs-NVMe-NUMA configurations on energy consumption of the processors?

Based on our experimental results, we develop a performance-energy analytical model to project the I/O performance and energy consumption of modern NVMe-based systems. The results are used to guide the design of the proposed scheduling algorithm for I/O requests called *Energy-efficient Scheduler for NVMe* or ESN. ESN can help improve the energy efficiency by judiciously mapping I/O threads to the appropriate CPU nodes without decreasing the throughput and latency. In summary, our work makes the following contributions.

- We demonstrate that utilizing more than two CPUs in the NUMA architecture can actually degrade the I/O performance. We find that using a single CPU achieves the best performance when the number of parallel I/O threads is less than 512, and utilizing two CPUs performs best when a larger number of threads are used. When more than two sockets are used, we see degradation of both I/O performance and energy efficiency.
- We implement a profiling-based user-level scheduler for parallel I/O threads, ESN, which achieves optimal throughput performances and small latencies, while consuming 50% less energy than the existing scheduler. We also propose a self-adjusting energy-efficient I/O threads mapping scheduler.

The rest of this paper is organized as follows. Section II introduces the background information of this work. Section III explains the software and hardware platforms used in this work. Section IV demonstrates the experimental evaluation results of the NVMe SSDs attached to the NUMA architecture, from which useful observations are drawn. Section V presents the ESN scheduler, along with an evaluation of ESN. Section VI discusses the design of a self-adjusting energy-efficient I/O scheduler. The related works are introduced and discussed in Section VII. Section VIII concludes this work.

II. Background

This section provides the background information on the current implementation of the I/O path in Linux 4.0.0 or later, the NVMe connections on NUMA processors, and the impact of process scheduling on the parallel I/O threads performance.

A. I/O Path

Multi-queue block layer and NVMe improve the concurrent I/O accesses by reducing lock contention through employing more I/O job queues implemented in both the software and hardware layers [1], [2]. Each CPU maintains its own I/O request queue (software staging queue) containing requests from the CPU. Each device also has one or more hardware dispatch queues for the I/Os. If a device is connected with NVMe, the hardware dispatch queues in the block layer are used as submission and completion queues in NVMe. On the contrary, the older implementation of Linux only has one queue for all I/O requests in both software and hardware layers.

There are two other types of contention for the shared resources that can reduce I/O throughput and increase latency. The first is the contention for the NVMe device. Even when both the device and connection support the parallel execution, the physical limitation of the NVMe and the bandwidth of the NVMe cannot fulfill all I/O requests if there are too many requests at the same time. The second type is the contention for the CPU by multiple I/O requests. The block I/O path uses polling instead of interrupt, as NVMe devices are fast. However, this can result in more CPU usage due to polling. When multiple I/O processes are running on the same CPU, these I/Os can contend, resulting high polling time.

B. NUMA Effect

Because each device is attached to the CPU directly via the NVMe in a similar fashion to that of DRAM modules, there are local and remote accesses to the device from different CPU nodes in a NUMA system as shown in Figure 1. A local access occurs when the data is available on the device module connected to a CPU that performs that I/O access. A remote access occurs when a CPU node tries to operate data on a device attached to another CPU. The multi-queue block layer prioritizes local I/O accesses. However, once there are more I/O requests than the number of the processing cores on a local CPU, the OS distributes some I/O requests to other available cores, resulting in more remote NVMe accesses.

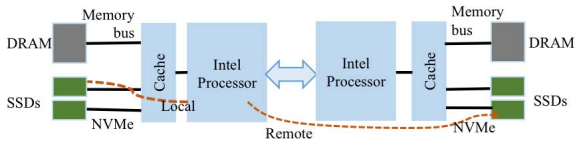


Fig. 1: Local/Remote NVMe SSD access on NUMA

The throughput and latency are different for local and remote NVMe SSD accesses, as shown in Table I. The data is based on our experimental measurements made with

ioping [17]. Remote accesses take longer because they need additional interconnect traversals and access contention. This behavior is the same as that of remote memory accesses in a typical NUMA system.

	Local	Remote
Bandwidth (GB/sec)	2.2	1.2
Latency (us)	10.9	21.8

TABLE I: Performance of local and remote accesses

C. Process Scheduling

In addition to the cost of remote access, CPU contention can also degrade I/O performance. Each I/O thread is mapped as a kernel process in Linux and is subjected to OS process scheduling. Because of the locality and the contention for shared CPUs and other resources, employing fewer threads on a single CPU often results in better performance. However, assigning fewer threads to a CPU can result in more remote accesses; therefore, it is possible that the cost of these remote accesses would offset the gain from fewer instances of CPU contention. Due to such interplays between remote access and CPU contention, it is also important to understand the I/O performance degradation caused by the process contention.

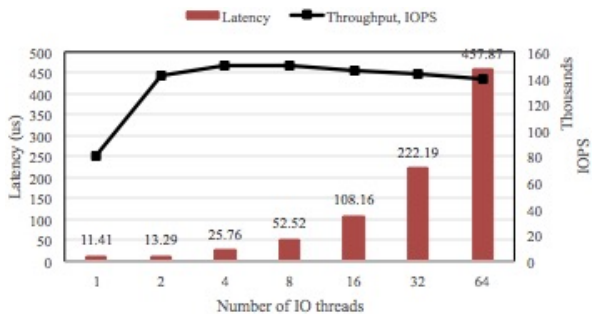


Fig. 2: I/O performance with respect to I/O threads running on a single CPU

Figure 2 presents changes in I/O performance when more I/O threads are running on a single CPU. The data shows that the throughput increases with the number of I/O threads up to a limit. The latency also increases with more I/O threads due to CPU contention. Considering the trade-off between the throughput and the latency, the optimum combination appears to be 2 I/O threads running on the same core concurrently.

CPU contention can occur only if there are more I/O threads than available cores. When the contention does occur, however, it is not clear which strategy is better: distributing the excess threads to remote CPUs to reduce CPU contention or keeping the threads on the local node to avoid the remote-access penalty. The answer depends on the relative costs of CPU contention vs remote access. Clearly, a performance model is called for to resolve this issue. The model should, additionally, consider energy consumption, because the energy consumption will rise as more CPUs are deployed.

D. Energy Consumption

Energy is consumed by various components within a system including NVMe devices, memory systems, peripherals, interconnects, and CPUs. For multi-core and many-core systems, studies have shown that CPUs consume the most power [18], [19]. Each CPU in our experimental platform has three states: busy (normal and low power), idling, and shutdown. According to its specification [20], the power consumption of different states in an Intel Xeon processor are compared in Table II.

The Intel processor adopts Dynamic Voltage and Frequency Scaling (DVFS) to adjust power consumption, which means that the actual energy consumption in a given period is determined by the workload of the processor as well as its state, as shown in Table II [21]. In order to accurately estimate the processor power consumed by the I/Os in different CPU states, we used the Intel power gadget tool [22] to compile the data.

CPU state	Busy		Idling (C3)	Shutdown (C6)
	Normal (C0)	Low power (C1)		
Power Specifications (W)	95	47	22	14

TABLE II: Power specifications of CPUs' different states

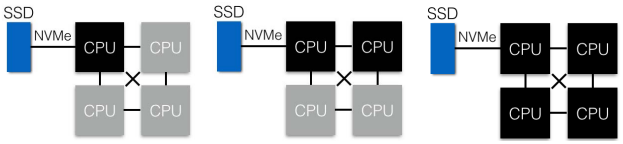
III. Experiment Methodology

Experimental Platform. Our experimental platform is a 4-socket Intel E5-4610 v2, with each socket having 8 physical cores (with hyper-threading) and 16MB last-level cache. This particular architecture uses the Sandy Bridge technology, which attaches the PCIes to different NUMA sockets, respectively [20]. The NVMe SSD device is Intel SSD DC P3700. There are 2 NVMe SSDs installed in the system, and each can be attached to one CPU socket with 4 PCIe lanes. The PCIe bandwidth is 1GB/sec. We use Linux kernel version 4.0.0, equipped with multi-queue block layer implementation and latest NVMe driver.

Item	Version/Count
Processor	Intel E5-4610 v2, 16M Cache
# of sockets	4
Cores per socket	8
Memory	256 GB
NVMe SSD	2*Intel DC P3700
Linux kernel	4.0.0-rc7 (blk-mq enabled)
FIO	2.2.8

TABLE III: Experiment platform characteristics

Software Systems. The I/O workload is generated and tested with flexible I/O generator (*fio*-2.2.8) [23]. By default, the Linux OS uses page cache for all the I/Os to improve the I/O performance. It stores the disk page data in main memory for quicker I/O accesses. In our experiments, however, we leveraged the *fio*'s direct I/O feature to bypass the page cache. Again, this is because NVMe SSD operates at a much higher speed than typical I/O devices. The I/O workloads are generated by *fio*'s `libaio` engine. The performance measurements, including the submission latency, completion



(a) Config. 1 (b) Config. 2 (c) Config. 3

Fig. 3: Experimental configurations of I/O mappings for 1 NVMe SSD. Config. 1, all I/O threads on local CPU socket. Config. 2, all I/O threads on local and nearest CPU sockets. Config. 3, all I/O threads on all 4 CPU sockets

latency, throughput (bandwidth) and disk utility, are reported by *fio*. Unless stated otherwise, the I/O block size is 4KB which is recommended as the best configuration in the NVMe SSD's specification document [24]. Intel power gadget [22], a software-based tool to monitor power usage of Intel processors, is adopted to measure the energy consumed during our experiments. To map the I/O threads to different CPUs and CPU sockets, *numactl* [25] is adopted.

Performance metrics. We evaluate the I/O performance in terms of the throughput in IOPS and latency in microseconds (μ-seconds). The energy consumption is measured in milliwatt hour (mWh).

IV. Experimental Results

In this section, we conducted a set of experiments to observe the impacts of various possible runtime penalties by doing our best to isolate each of the potential factors. We use NUMA architecture for this experiment and explored different CPU configurations to evaluate factors such as resource contention and remote memory access penalties, that can degrade performance. Second, because I/O threads are also subjected to scheduling by the default Linux scheduler, we also investigated the impacts of process scheduling on I/O performance. Our investigation was done on the two best performing configurations. Lastly, we discuss a dilemma of achieving optimal I/O performance while conserving energy.

A. Effect of Process Scheduling in a Single NVMe SSD System

So far, the I/O performance is evaluated with the Completely Fair Scheduler (CFS), the default scheduler in Linux [26]. However, the NUMA architecture introduces the remote device access penalty, on top of the shared resource contention between I/O threads. Process-contention penalty and remote-device-access penalty are two possible I/O performance bottlenecks that can significantly degrade the overall performance [27]. In this section, we investigate the impacts of I/O scheduling on these two types of penalties and how it can affect the performance and scalability of NVMe SSD.

We mapped I/O threads to different CPU sockets to show the trade-offs between these two penalties. Figure 2 in Section II demonstrates the effects on performance due to increased contention between multiple I/O threads on a single CPU. Furthermore, the throughput and latency of parallel I/O threads requesting from a single NVMe SSD are evaluated with three different NUMA configurations as shown in Figure 3: 1) all I/O threads running on only a local CPU socket, 2) all I/O threads running on a local and a nearest neighboring remote socket, and 3) all I/O threads running on all four sockets, which is referred to as the *default setting* henceforth. The CFS equally distributes all I/O threads to the available CPU cores to balance the CPU workload [26]. Figure 4 presents the impacts of contention and remote NVMe SSD access penalty on the performance of parallel I/O threads. As shown, there are two interesting numbers of I/O threads, 16 and 512 that show significant changes in performance.

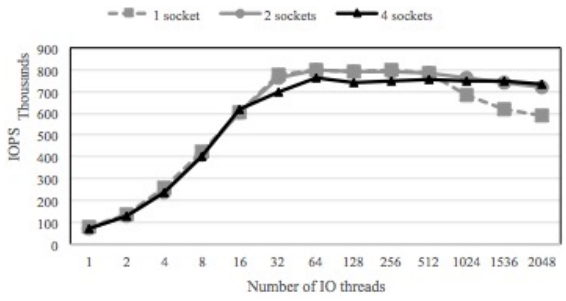
When the number of I/O threads is less than 16, there is no difference in the number of available CPU cores among the three mapping configurations. This is because all I/O threads can run on a local CPU socket automatically. Although there are other CPU sockets available in the second and third configurations, only CPU cores on the local socket are busy running. The throughput is the same for all configurations, but the latency is best when only the local CPU socket is used. In the second and third configurations, the CPU cores on remote sockets can respond to I/O interrupts. The OS may also migrate I/O threads to the local CPU node for better locality. Such response latency and migration cost can prolong the I/O latency.

When the number of I/O threads is between 16 and 512, both throughput and latency are worst when four CPU sockets are used among the three configurations. This is because of the remote access penalty on NUMA, as some I/Os are running on the remote CPU sockets. Both I/O throughput and latency are the same with one and two CPU sockets, which is reasonable according to Figure 2, which shows that CPU contention also degrades the I/O performance. This indicates that the contention penalty and remote access penalty have similar effect on the I/O performance when the number of I/Os falls into this range.

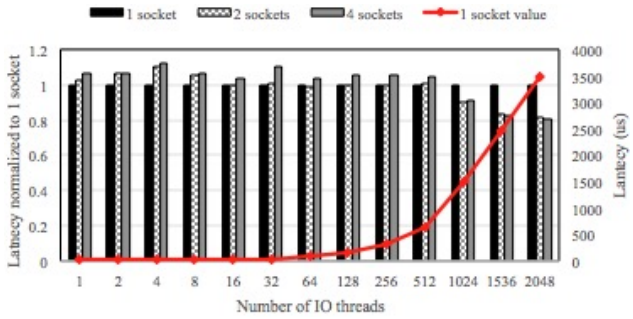
When the number of I/Os is more than 512, the performance with one CPU socket is worst but the other two settings show similar throughputs and latencies. The contention among these 512 I/O threads has more negative impacts on I/O performance than those of the remote device access penalty. With two CPU sockets the contention is less than that of only one CPU socket.

According to the reported results, we observe that when the number of I/O threads reaches 1536, the configuration with four CPU sockets delivers the best I/O performance. When we have more than 512 I/O threads per CPU socket, it is better to distribute additional I/Os to neighboring CPU sockets.

An important point to consider based on the above observation is that when we utilize fewer CPUs, the system also uses less energy to operate. If the same I/O performance can be achieved with fewer CPUs, it is better to map the I/Os to a set of CPUs and then idle or turn off the remaining CPUs. With one NVMe SSD, only the local CPU socket is the best in terms of energy efficiency and I/O performance when there are 16 I/O threads or less. The configuration with one local and one nearest neighboring CPU sockets performs best in



(a) Throughput



(b) Latency

Fig. 4: Comparison of I/O performance for one NVMe SSD when all I/O threads are allowed to run on one, two or four NUMA nodes with 1 to 2048 I/O threads.

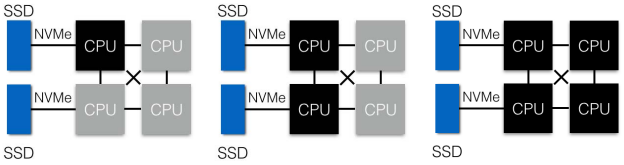
terms of I/O latency, I/O throughput and energy consumption in the remaining scenarios.

B. Effect of Process Scheduling in a Two NVMe SSDs System

To show the limitation of bandwidth and its effects, we conducted experiments using two NVMe SSDs with two different types of connections: 1) two NVMe SSDs are connected to two CPU sockets, and 2) two NVMe SSDs are connected to the same CPU socket.

1) Two NVMe SSDs Connected to Two CPU sockets: In this configuration, each NVMe SSD has its own local CPU socket as shown in Figure 1. The workload and number of parallel I/Os for each NVMe SSD is the same, but the order in which read/write requests are directed to each device is random. Three NUMA mapping configurations are also adopted as shown in Figure 5: 1) all I/O threads run on one local CPU socket (randomly choosing one NVMe SSD); 2) the I/O threads run on two local CPU sockets each connected to a NVMe SSD; and 3) the default setting of having all I/O threads run on all four sockets.

The experiment results are presented in Figure 6. Configuration 1) performs the worst as half of the I/O requests are issued to one remote NVMe SSD. There are little differences in the throughput performances between configuration 2) and 3), when the number of I/Os is less than 512. This is consistent with the results of a single NVMe SSD where mapping fewer than 512 parallel I/Os to local socket gets the best I/O performance.



(a) Config. 1 (b) Config. 2 (c) Config. 3

Fig. 5: Experimental configurations of I/O mappings for 2 NVMe SSDs that are connected to different CPU sockets. Config. 1, all I/O threads on one local CPU socket. Config. 2, all I/O threads on two local CPU sockets for each SSD. Config. 3, all I/O threads on all 4 CPU sockets.

The latency is worst in the configuration with one CPU socket (Configuration 1) than that of Configuration 3. As a reminder, this is due to the initial CPU response latency and I/O migration cost. With more I/O threads, the latencies between Configurations 2 and 3 become similar because in Configuration 2, I/O threads contend for CPU more.

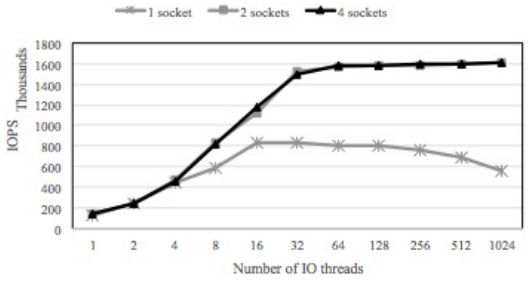
In summary, we observe that Configuration 3 suffers from CPU response latency and I/O migration cost. At the same time, the Configuration 2 also suffers from the contention penalty. As such, these two systems show nearly identical throughput performance as shown in Figure 6(a), and very little difference on latency performance as shown in Figure 6(b).

When there are more than 512 I/O threads, the penalty due to contention is larger than that due to remote access in the case of using just one CPU socket. To reduce the contention penalty, we utilize neighboring CPU sockets to support additional I/O threads. As shown in Figure 6, the latency with 4 CPU sockets is shortest but the throughput performances between configuration 2) and 3) are similar.

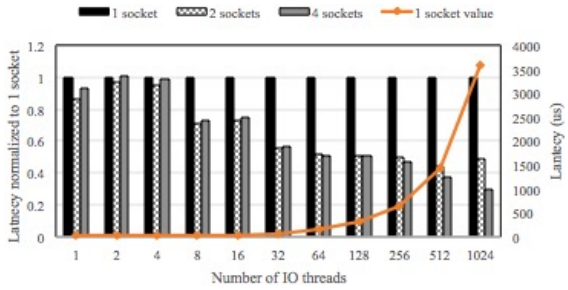
2) NVMe SSDs share one CPU socket: Each CPU socket in our platform has 40 PCI-e lanes that can host up to 10 NVMe modules. We experimented with another configuration that connects two NVMe SSDs to one CPU socket; i.e., these two NVMe SSDs share the same local CPU socket. Three NUMA mapping configurations are adopted as shown in Figure 7: 1) all I/Os run on only the local CPU socket; 2) the I/Os run on the local and a nearest neighboring CPU sockets; and 3) all I/Os run on all four sockets, which is the default setting.

Figure 8 demonstrates the I/O performance with different numbers of I/O threads. Due to additional contention for the shared CPU and buses, mapping all I/O threads to a local CPU socket performs worse than the other two configurations. Using two CPU sockets (local and one nearest neighboring sockets) has the same I/O performance as the default configuration, which uses four CPUs.

Summary. We observe that we can use fewer CPUs to deliver the same I/O performance using all four CPUs. This means that if we should use a configuration that utilizes fewer CPUs , we can conserve energy while maintaining the optimal performance. Next, we present a model that can provide guidance on how to configure the system to strike a good balance between performance and energy conservation.

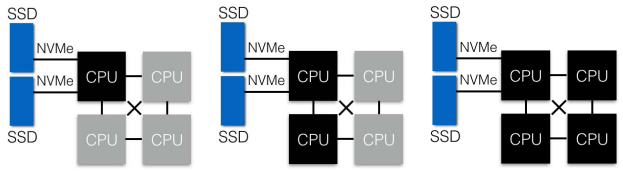


(a) Throughput



(b) Latency

Fig. 6: Comparison of I/O performance for two NVMe SSDs (each connected to different NUMA node) when all I/O threads are allowed to run on one, two or four NUMA nodes, the number of I/O threads changes from 1 to 2048. The xlabel is the number of I/O threads to each NVMe SSD, ranges from 1 to 1024.



(a) Config. 1 (b) Config. 2 (c) Config. 3

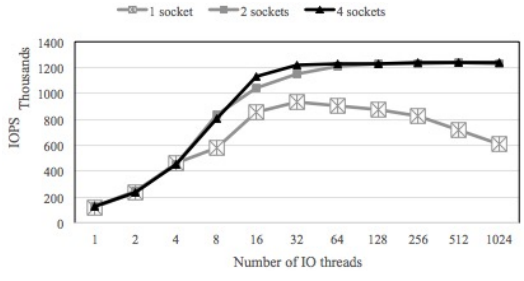
Fig. 7: Experimental configurations of I/O mappings for 2 NVMe SSDs that are connected to same CPU sockets. Config. 1, all I/O threads on local CPU socket. Config. 2, all I/O threads on local and nearest CPU sockets. Config. 3, all I/O threads on all 4 CPU sockets

V. ESN: Energy-efficient I/O Scheduler

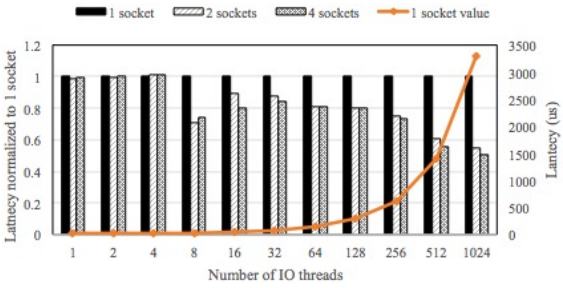
Based on our evaluation results, we implemented ESN, an I/O thread scheduler that can help determine how to map I/O threads to the CPUs at runtime to achieve the best I/O performance and the least energy consumption for a given I/O workload. ESN is a profiling-based, user-level runtime scheduler, which makes the mapping decision during the execution according to the runtime environment, underlying platform and the the locality of the I/O threads.

A. Design

ESN has been designed to reduce energy consumption without sacrificing the I/O performance. Algorithm 1 describes the



(a) Throughput



(b) Latency

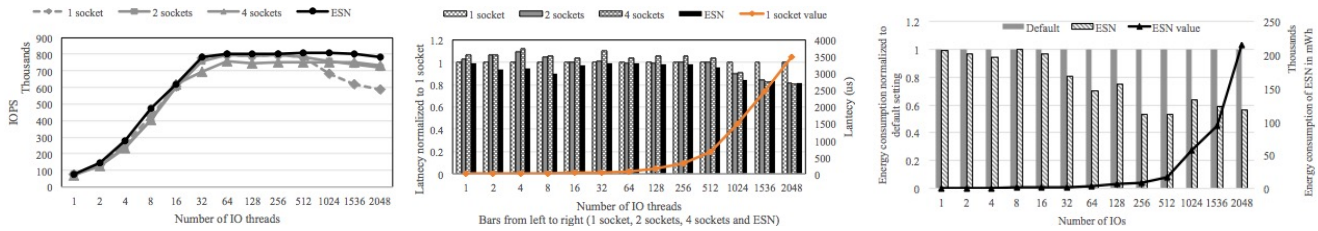
Fig. 8: Comparison of I/O performance for two NVMe SSDs (both connected to same NUMA node) when all I/O threads are allowed to run on one, two or four NUMA nodes. The xlabel is the number of I/O threads to each NVMe SSD, ranges from 1 to 1024.

proposed energy-efficient I/O scheduler for multiple NVMe SSDs on a NUMA system. The scheduler first identifies the number of I/O target devices, and then counts the number of concurrently running processes on the local CPU socket, N_{local}, by reading system file */proc/$pid/stat*. It then compares N_{local} with N to determine if the nearest neighboring CPU socket should be used to host the I/Os. In other words, *N is the number of parallel IO threads running on the local socket that would cause the contention penalty to be greater than the remote access penalty*.

Based on the previously presented observation, the contention penalty has less negative effect than the remote access penalty. As such, ESN detects an increase in the numbers of I/O threads, which leads to an increasing contention penalty, and then allocates these I/O threads to the nearest neighboring CPU once the number of I/O threads is more than the number N. Currently, a profile run is needed to generate the optimal schedule. We leave an automatic approach to generate an optimal schedule as future work.

The criterion to decide N is identifying the number of parallel I/O threads running on the local socket that would cause the contention penalty to be larger than the remote access penalty. As shown previously, N is 512 on the platform used in this study.

It must be noted that the various parameters and coefficients in the proposed performance-energy model can be obtained for a different platform and/or NVM(SSD)-NVMe-NUMA configuration by a combination of profiling and curve-fitting



(a) Throughput (b) Latency (c) Energy

Fig. 9: Comparison of I/O performance for one NVMe SSD when all I/Os are scheduled by ESN and allowed to run on one, two or four NUMA nodes. Xlabel is the number of parallel I/Os to one NVMe SSD, ranges from 1 to 1024.

Algorithm 1: The proposed energy-efficient scheduler

```
Initialization: launch the I/O application (A) and the
helper script (S) concurrently;
for each thread (T) in A do
    count the number (N_local) of processes running on
     local socket;
    if N_local smaller than number N then
        map T to local CPU socket (or certain CPU
         cores);
    else
        map T to neighbor remote CPU socket;
    end
end
```

processes as explained in Section IV.

B. Evaluation

We compare the I/O performance and energy consumption of a system using the proposed ESN scheduler to that of a system using the default Linux I/O scheduler. We previously described the experimental platform in Section III.

One NVMe SSD. Figure 9 presents the performances of the proposed ESN and those of the default Linux I/O scheduler using different CPU settings. As shown, the throughputs and latencies are nearly identical to the settings using 1 or 2 NUMA sockets. When there are more than 1024 I/O threads, the proposed scheduler performs better than the other settings because ESN allocates two nearest neighbor sockets compared with the two-socket setting that uses only one nearest neighbor socket (more contentions) and default four sockets setting that has higher remote access penalty.

The energy consumed by the CPUs using ESN is normalized to the energy consumed by CPUs using the default setting. Because direct I/O is used in our experiment, the profiling results indicate similar DRAM energy consumption profiles for both schedulers. The profiling results also indicate that when there are fewer than 16 I/O threads, ESN consumes the same amount of energy as that of the Linux I/O scheduler. This is because ESN, in effect, binds threads to the CPUs in the same fashion as the Linux I/O scheduler. However, when more I/O threads are used, ESN consumes less energy than the default scheduler while maintaining a similar I/O performance. Because ESN introduces more contention, it incurs more workload on the CPUs than the default setting. In addition, the default scheduler also utilizes more CPUs so each CPU is less busy. As such, the energy saving of ESN is not proportional to the number of CPU sockets used by the default scheduler.

Two NVMe SSDs connected to two NUMA sockets. Figure 10 compares the performance of ESN with that of the default scheduler. As shown, the performances of ESN and the default scheduler are the same but ESN consumes less energy. Because I/O requests are issued independently to each NVM, the I/O threads are then mapped to the local CPUs for each NVM. When there are fewer than 512 I/O threads for each NVM, all threads run on the local sockets of each NVM. The nearest neighbor socket is used if the number of I/Os is more than 512. When there are more than 768 for each NVMe SSD, all four sockets would be used to run the I/O threads with more bias toward running them on the two local CPU sockets.

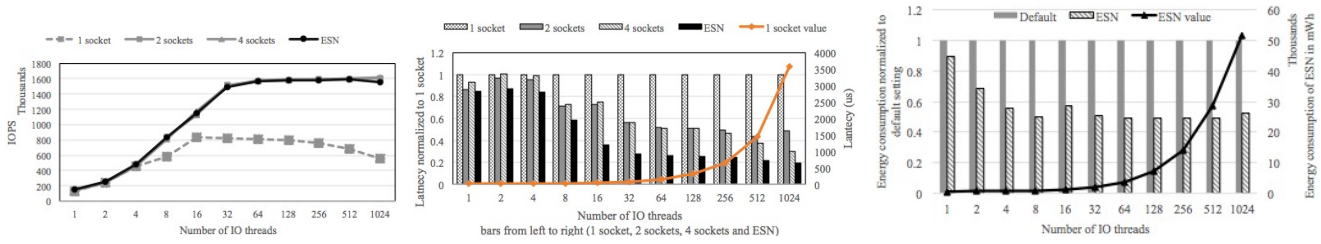
Different from the one NVMe SSD energy comparison result, ESN performs better for two NVMe SSDs from the beginning. ESN consumes less energy than that of the default scheduling because, *first*, ESN avoids thread migration costs by mapping the I/O threads to the local CPU socket of each NVMe SSD from the time they start; *second*, ESN uses only half of the CPUs than default approach.

C. Discussion

The tradeoff between I/O throughput, latency, and energy consumption is described in Equation 1. In this equation, a higher performance tradeoff value implies higher throughput, shorter latency, and less energy consumption. In the equation, three elements are equally weighted, but the equation can be modified so that elements are weighted differently. For example, to design a system focused on shorter latency, we can replace $Latency$ in the equation with $Latency^2$ to increase the weight.

$$Performance_{tradeoff} = \frac{Throughput}{Latency \times Energy} \quad (1)$$

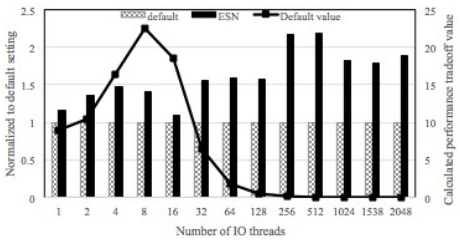
Figure 11 compares the performance of ESN with that of the default scheduler for one and two NVMs according to



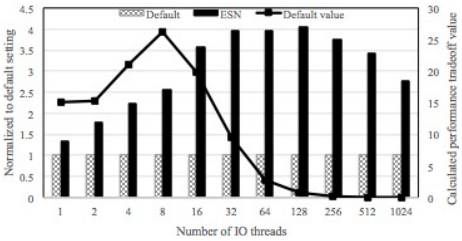
(a) Throughput (b) Latency (c) Energy

Fig. 10: Comparison of I/O performance for two NVMe SSDs when all I/Os are scheduled by ESN and allowed to run on one, two or four NUMA nodes. Xlabel is the number of parallel I/Os to each NVMe SSD, ranges from 1 to 1024.

Equation 1. The calculated values of ESN is normalized to the corresponding value of the default scheduler while managing the same number of parallel I/O threads.



(a) 1 NVMe SSD performance tradeoff



(b) 2 NVMe SSDs performance tradeoff, x-axis is the number of I/Os issued to each NVM

Fig. 11: Performance tradeoff comparison between ESN and default (4 sockets). Bars are values normalized to default setting's value (higher better). Line is the calculated value of default setting based on Equation 1.

Figure 11 (a) shows the calculated values of one NVMe SSD. The best performance is achieved with 8 I/O threads (i.e., at 8 I/O threads, the actual magnitude as shown using the black line is the highest). This conclusion is based on the combination of throughput, latency and energy. When there are more I/O threads, as illustrated in Section IV, the energy consumption also increases with the improved throughput and reduced latency. The optimal number of I/O threads, 8, is also the number of physical processing cores on a local NUMA socket.

When there are two NVMe SSDs, the combined performance continues to improve with more I/O threads (i.e., up to 1024 I/O threads) according to Figure 11 (b). This is true because the throughput is twice as large while the latency is half as small. Processors and NVMe also experience less contention.

VI. Self-Adjusting Energy-efficient I/O Scheduler

In this section, we extend ESN to make it self-adjusting. We conducted our experiment on a simulator for parallel I/Os to NUMA-based NVMe devices to demonstrate the flexibility and effectiveness of the proposed scheduler on a larger platform that has more sockets and cores than our previous system. We used SPIN[1] to simulate the parallel I/Os to NUMA-based NVMe storage devices. The simulated platform has 6 CPU sockets and each socket has 8 cores (i.e., two more sockets that our actual system used in Section IV). The results are then compared against those of the default CFS scheduler.

As demonstrated in Section IV, the I/O performance and energy consumption are better when I/O processes running on local CPU socket until the contention penalty is worse than NUMA penalty. The design of the proposed self-adjusting scheduler is based on the analysis of NUMA and contention penalties. Because both penalties depend on the platform and its operation [28], the penalty values adopt in this scheduler are the results reported in prior research. For example, we use the prior reported results indicating that NUMA-penalty can result in 30% performance degradation and contention-penalty is a second polynomial order of the number of the I/O processes [29].

To obtain accurate penalty values, one approach is using profile runs as explained in Section IV and Section V. Another approach is using a self-adjusting process to tune the penalty values during the runs. This approach suffers from poor performance during the initial runs but achieves good performance improvement in the subsequent runs. In this study, we take the self-adjusting approach.

Algorithm 2 describes the proposed self-adjusting energy-efficient I/O scheduler. Before scheduling the I/O processes, the algorithm identifies the platform that the local/neighbor CPU sockets of the NVMe device and the number of I/O processes on local CPU socket to achieve best I/O and energy performance with Equation 2, where N is the number of I/O processes and α: -1936, β: 628201, γ: 9829.

[1]https://github.com/junjieqian/spin

$$\begin{cases} NUMA - penalty = 1.3 \\ Contention - penalty = \alpha * N^2 + \beta * N + \gamma \end{cases} \quad (2)$$

Algorithm 2: The proposed energy-efficient scheduler

Platform identification: Identify the number of CPU cores per socket (N_{cpus}), the local\nearest_neighbor\remote CPU sockets;
Pre-scheduling: Calculate the balance point (the number of I/O processes per local CPU socket, $N_{I/O}$) that the contention penalty equals the NUMA penalty;
Initialization: Launch the I/O application (A) and the helper script (S) concurrently;
Scheduling: 1. Identify the number of existing I/O processes on local CPU socket, $N_{existing}$;
if $N_{existing}$ *is less than* $N_{I/O}$ **then**
    Map part of the I/O processes in A to local CPU socket, $N_{I/O} - N_{existing}$;
    while *not all I/O processes in A are mapped* **do**
        Map $n * N_{I/O}$ of the I/O processes in A to n adjacent neighbor CPU sockets;
    end
else
    while *not all I/O processes in A are mapped* **do**
        Map $n * N_{I/O}$ of the I/O processes in A to n adjacent neighbor CPU sockets;
    end
end
Periodically re-balance the load when one or more I/O processes on local CPU socket finish.

As previously mentioned, we used SPIN to simulate the hardware system running our self-adjusting scheduler. Table IV compares the performance of the proposed scheduler to that of the default CFS I/O scheduler. As shown in the table, the proposed self-adjusting scheduler can achieve the same throughput performance as that of CFS in most cases with significantly less energy consumption and shorter latency. As the table reports, in the case of 1024 parallel I/O threads, we can reduce latency by 70% and energy consumption by 25%, while retaining 90% of the throughput performance.

VII. Related Work

In this section, related research efforts on NVMs and I/O schedulers for SSDs are presented.

NVM studies can be broadly categorized into performance analysis, performance optimization, and applications of NVM modules as memory. Work by Zhang et al. [5] and Sehgal et.al [10] use the PMEP emulator [12] to emulate part of DRAM as NVM and investigate file system performances on NVMs. Son et al. [7] optimize the file system for fast storage device and the battery-backed DRAM is used as NVM. Vuvcinic et al. [6] explore the performance bottlenecks of PCI-e for NVMs and implement a new protocol for much higher I/OPS. Awad et al. [8] analyze the performance impacts of PCI-e connections and design a new interface for better parallelism and low-latency. These studies are based on simulation or emulation, while we are using real NVM devices for evaluations.

	32 I/Os	128 I/Os	512 I/Os	1024 I/Os
Throughput	1	1	1	0.9
Latency	0.625	0.7	0.7	0.3
Energy	0.6	0.3	0.67	0.75

TABLE IV: Performance of the proposed scheduler for 1 NVM compared with CFS scheduler.

Xu et al. [9] investigate how different databases perform with NVM device but only use the single queue block layer which does not reflect the impact of the latest I/O path in the software stack. Onagi et al. [30] evaluate the effect of wear leveling on the PCI-e connected SSD performance, energy consumption, and endurance. Cully et al. [31] present a storage system that has two parts, an address virtualization layer for network attached PCI-e connected SSDs and a host environment for scalable protocols.

I/O schedulers are mostly designed based on an assumption that most storage devices are slow, and therefore, a single queue block layer is acceptable. However, there have been work to conserve energy by creating custom I/O schedulers specific for SSDs.

Cheng et al. [13] analyze the energy consumption of I/Os in hard real-time systems with a preemptive periodic task model and propose an online scheduler that utilizes device slack to perform power state transitions for better energy performance. Ge et al. [15] propose an I/O scheduler for data intensive applications, which changes the voltage and frequency of processors to save the energy. Gim et al. [32] propose a mechanism to decide if an I/O request should be serviced using interrupt or pooling according to device and other aspects such as CPU utilization and I/O size. Akram et al. [33] studied the NUMA effects on storage I/O performance. Their paper presents that remote NUMA node access degrades the I/O performance as well, but without solution and investigations on the energy consumption.

Energy aware schedulers have been recently investigated. For example, Ali et al. [34] present a power aware scheduler on the NUMA architecture for hypervisor. Frasca et al. [19] propose a NUMA aware graph mining technique to improve both performance and energy consumption. Gough et al. [18] investigate power management of memory system and I/Os in servers. Qian et al. [35], [36] study energy efficient schedulers in mobile devices. However, these techniques do not address energy consumption issues in NVM-based NUMA systems.

VIII. Conclusion

In this work, we show that it is possible to achieve good parallel I/O performance while reducing energy consumption of NVMe-based NUMA systems. Our insight is obtained through an in-depth investigation of various penalties due to resource contention and remote NVMe SSDs accesses in NUMA. We find that contention among I/O threads on shared resources such as CPUs incurs less penalty than that

incurred by remote NVMe device access in NUMA. Based on this insight, we implement an energy-efficient profiling-based I/O scheduler, called ESN. ESN maps the I/O threads to local or nearest neighbor remote CPU sockets according to the number of concurrently running I/O threads on a CPU socket. As compared to the default scheduler, ESN can deliver equivalent parallel I/O performance, while consuming up to 50% less energy by idling CPUs that cannot contribute to better throughput performance.

ACKNOWLEDGMENT

Part of this work is done by Junjie Qian during his 2015 Summer internship in NetApp Inc. This material is based on research sponsored by NetApp Inc, NSF and Maryland Procurement Office under agreement number CNS-1116606, CCF-1629625 and H98230-14-C-0140. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. Government.

REFERENCES

[1] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: introducing multi-queue SSD access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 22.

[2] A. Huffman and D. Juenemann, "The nonvolatile memory transformation of client storage," *Computer*, no. 8, pp. 38–44, 2013.

[3] J. Min, S. Ahn, K. La, W. Chang, and J. Kim, "Cgroup++: Enhancing I/O Resource Management of Linux Cgroup on NUMA Systems with NVMe SSDs," in *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference*. ACM, 2015, p. 7.

[4] "Non-Volatile Memory Express," http://www.nvmexpress.org/.

[5] Y. Zhang and S. Swanson, "A Study of Application Performance with Non-Volatile Main Memory," in *Mass Storage Systems and Technologies (MSST), 2015 31th Symposium on*. IEEE, 2015, pp. 1–10.

[6] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. Le Moal, T. Bunker, J. Xu, S. Swanson *et al.*, "DC express: shortest latency protocol for reading phase change memory over PCI express," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 2014, pp. 309–315.

[7] Y. Son, H. Han, and H. Y. Yeom, "Optimizing file systems for fast storage devices," in *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, 2015, p. 8.

[8] A. Awad, B. Kettering, and Y. Solihin, "Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems," in *Performance Analysis of Systems and Software, 2015 IEEE International Symposium on*. IEEE, 2015, pp. 145–154.

[9] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and their Implication on Real World Databases," in *Proceedings of the 8th International Systems and Storage Conference*. ACM, 2015, p. 22.

[10] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An Empirical Study of File Systems on NVM," in *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST15)*, 2015.

[11] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A Lightweight Performance Emulator for Persistent Memory Software," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 37–49.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15.

[13] H. Cheng and S. Goddard, "Online energy-aware I/O device scheduling for hard real-time systems," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006, pp. 1055–1060.

[14] W. Sul, H. Eom, and H. Y. Yeom, "Energy-Aware I/O Scheduler for Flash Drives," in *Information Science and Applications (ICISA), 2014 International Conference on*. IEEE, 2014, pp. 1–4.

[15] R. Ge, X. Feng, and X.-H. Sun, "SERA-IO: Integrating energy consciousness into parallel I/O middleware," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 204–211.

[16] S. Park, W. Jiang, Y. Zhou, and S. Adve, "Managing Energy-performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. ACM, 2007, pp. 169–180.

[17] "ioping: simple disk i/0 latency measuring tool," https://github.com/koct9i/ioping.

[18] C. Gough, I. Steiner, and W. Saunders, "Memory and I/O Power Management," in *Energy Efficient Servers*. Springer, 2015, pp. 71–91.

[19] M. Frasca, K. Madduri, and P. Raghavan, "NUMA-aware graph mining techniques for performance and energy efficiency," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–11.

[20] "Intel Xeon Processor E5-2600/4600 Product Family Technical Overview," http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-1.pdf.

[21] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An Energy Efficiency Feature Survey of the Intel Haswell Processor," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), IEEE International*. IEEE, 2015.

[22] "Intel Power Gadget," https://software.intel.com/en-us/articles/intel-power-gadget-20.

[23] "Fio: Flexible I/O Tester Synthetic Benchmark," https://github.com/axboe/fio.

[24] "Intel SSD DC P3700 Series Specifications," http://www.intel.com/content/www/us/en/solid-state-drives/ssd-dc-p3700-spec.html.

[25] "A NUMA api for Linux," http://linux.die.net/man/8/numactl.

[26] "Completely Fair Scheduler in Linux," https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, 2015.

[27] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on NUMA systems: asymmetry matters," in *Proc. of the USENIX Conference on USENIX Annual Technical Conference*, 2015.

[28] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying numa and contention effects in multi-gpu systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 11.

[29] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," in *ACM SIGPLAN Notices*, vol. 46, no. 11. ACM, 2011, pp. 11–20.

[30] T. Onagi, C. Sun, and K. Takeuchi, "Design guidelines of storage class memory based solid-state drives to balance performance, power, endurance, and cost," in *Japanese Journal of Applied Physics*, vol. 54. IOP Publishing, 2015.

[31] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 2014, pp. 17–31.

[32] J. Gim, T. Hwang, Y. Won, and K. Kant, "SmartCon: Smart Context Switching for Fast Storage IO Devices," in *ACM Transactions on Storage (TOS)*, vol. 11. ACM, 2015, p. 5.

[33] S. Akram, M. Marazakis, and A. Bilas, "Numa implications for storage i/o throughput in modern servers," in *3rd Workshop on Computer Architecture and Operating System co-design (CAOS12)*, 2012.

[34] Q. Ali, H. Zheng, T. Mann, and R. Srinivasan, "Power Aware NUMA Scheduler in VMware's ESXi Hypervisor," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 193–202.

[35] H. Qian and D. Andresen, "An energy-saving task scheduler for mobile devices," in *Computer and Information Science (ICIS), 2015 IEEE/ACIS 14th International Conference on*. IEEE, 2015, pp. 423–430.

[36] ——, "Jade: Reducing energy consumption of android app," *the International Journal of Networked and Distributed Computing (IJNDC), Atlantis press*, vol. 3, no. 3, pp. 150–158, 2015.