Proceedings of 15th IOE Graduate Conference
Peer Reviewed
Year: 2024 Month: May Volume: 15
ISSN: 2350-8914 (Online), 2350-8906 (Print)

Enhancing NVMe Storage Performance with Latency-Aware User Layer Semantics and Dynamically Adjusted Timeouts

Manish Gyawali [a], Madhav Aryal [b]

[a, b] *Department of Electronics and Computer Engineering, Pulchowk Campus, IOE, Tribhuvan University, Nepal*
[a] 076msdsa007.manish@pcampus.edu.np, [b] 074bct520.madhav@pcampus.edu.np

Abstract
Upon completing a request, an I/O device faces the decision of either minimizing latency by promptly issuing an interrupt or optimizing throughput by delaying the interrupt, anticipating the completion of more requests soon and thus reducing the overall interrupt cost. To achieve a balance between these conflicting objectives, devices employ adaptive interrupt coalescing heuristics. However, these heuristics rely on static timeout and threshold values, leading to suboptimal performance in nonuniform IO workloads. Furthermore, devices lack semantic information regarding the latency sensitivity of I/O requests, which can result in undesired outcomes through interrupt coalescing. This paper proposes a method to enhance I/O device performance by enabling software to specify the latency sensitivity of requests. Subsequently, the kernel can utilize this information to dynamically assemble interrupts using adjusted timeout and threshold values. Additionally, we address the challenge of high interrupt rates and latency when all I/O requests are marked as latency-sensitive. We introduce a dynamic timeout value based on the remaining IO requests, considering the incoming IO request rate and the NVMe device's IOPS capability to optimize performance. Our approach Further eliminates the need for manual input of timeout and threshold values based on workload, instead utilizing an exponential-based correction factor. Experimental result shows that adding the user layer semantics in IO request using correction factor for timeout increases IOPS by 13% in comparison to interrupt coalescing techniques provided with current NVMe devices.

Keywords
NVMe, Performance, Coalescing, Kernel, IOPS, Interrupt

1. Introduction

In the evolution of data storage technologies, traditional disks once operated with seek times measured in milliseconds, accommodating at most a few hundred interrupts per second. This characteristic made interrupts a practical means to facilitate software-level concurrency while circumventing prohibitively high overheads. However, with the advent of modern storage devices that utilize solid-state memory, a paradigm shift has occurred. These advanced devices boast the capability not only to sustain millions of requests per second but also to handle multiple concurrent requests simultaneously [1].

Figure 1 provides a simple overview, which shows how NVMe requests are submitted and completed. It demonstrates how this storage framework efficiently manages parallel operations.

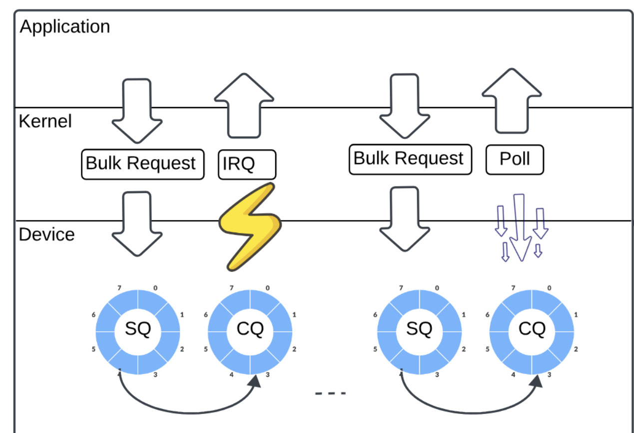

Figure 1: The problem of using original NVMe coalescing which can introduce high delay and timeout increasing latency.

A pivotal development in this context is the NVMe (Non-Volatile Memory Express) specification, which exposes the inherent parallelism of solid-state storage to software applications. NVMe achieves this by providing multiple queues, up to an impressive 64,000 per device, where requests, each queue supporting up to 64,000 requests, can be submitted and completed. This architectural enhancement has prompted Linux developers to adapt, leading to a substantial rewrite of the operating system's block subsystem to align with the multi-queue paradigm [2].

The ever-increasing performance of networking devices poses a challenge for storage systems. For example, a typical 100Gbps network card can process more than 100 million packets per second, far exceeding the capabilities of current storage devices like NVMe.

To address this challenge, the networking community has developed two main strategies: interrupt coalescing and polling[3]. Network devices use interrupt coalescing to prevent the CPU from overwhelming with constant interruptions. This technique groups a specific number of packets (threshold) or waits for a timeout before triggering an interrupt. Network stacks can also utilize polling, where software actively queries packets to process instead of relying on interrupt notifications. Additionally, technologies like

DPDK and DDIO bypass the kernel and interrupts entirely by exposing the device directly to applications in user space, further enhancing polling efficiency.

The NVMe specification attempts to adapt interrupt coalescing for storage devices, but it faces limitations. One limitation is the coarse-grained timeout. NVMe only allows one to set the aggregation time in 100 microsecond increments, which is unsuitable for devices achieving sub-10 microsecond latencies. This can significantly amplify the latency for small requests. Another limitation is the static configuration of both the threshold and timeout. These settings are fixed, making them ineffective for handling workload variations. Coalescing can easily break if the workload temporarily falls below the threshold.

Due to these limitations, the NVMe standard recommends disabling interrupt coalescing by default. Software and driver workarounds like interrupt completion threading and polling are still necessary to manage interrupt storms. Real-world deployments, like Azure, experience substantial latency increases with aggressive coalescing and require driver mitigation.

The ever-growing demand for data storage performance presents a challenge for traditional storage systems. New high-speed interconnects like NVMe allow applications to submit millions of requests per second, but this surge in I/O operations can lead to interrupt storms, overwhelming the CPU and grinding the system to a halt.

Interrupt coalescing, a common technique that batches requests into a single interrupt, has been explored to mitigate interrupt storms [4, 5, 6]. However, this approach relies on device-side heuristics that lack the semantic context to understand the intent behind each request, potentially leading to suboptimal performance.

This research extends a technique called "calibrating interrupts" (cinterrupts) [7] to address the challenge of managing exponentially increasing interrupt rates without compromising latency. cinterrupts bridge the semantic gap between hardware and software by adding two bits to requests sent to the device. This calibration information allows collaborative interrupt generation, ensuring timely completion delivery while avoiding interrupt storms. cinterrupts require minimal modifications to the device firmware but necessitate changes to the operating system.

The semantic information plays a crucial role in cinterrupts' effectiveness. The Linux kernel can automatically annotate I/O requests with default calibrations based on the system call that initiated the request [8]. In addition, an interface is introduced to allow applications to override these defaults for specific workloads [9].

Extensive evaluation through microbenchmarks showcases the efficacy of cinterrupts. It achieves performance comparable to state-of-the-art interrupt-driven approaches while consuming significantly fewer CPU cycles per request and improving throughput by up to 35% [10]. Furthermore, even without application-level modifications, cinterrupts utilizing default kernel calibrations demonstrate substantial performance improvements for prominent database systems like LevelDB and KVell. Benchmarks on uniform workloads show throughput increases of up to 14% and latency reductions of up to 28% compared to existing methods [10].

This paper extends Tai et al.'s [7] work on interrupt coalescing timeout by addressing its limitations with static delta values, particularly in variable workloads. In algorithm 1

Interrupts serve as a fundamental communication method between the operating system and devices. Although they enable concurrency and efficient completion delivery, associated costs, including context switches, are well documented[6, 11]. In storage, these costs have become particularly significant with the advent of new interconnects such as NVM ExpressTM (NVMe), which allows applications to submit millions of requests per second and handle up to 65,535 concurrent requests[12]. However, the sheer number of concurrent requests could lead to an interrupt storm, potentially overwhelming the system. Given that the CPU is already a bottleneck for achieving high IOPS, excessive interrupts could severely limit the ability of software to fully utilize current and future storage devices.

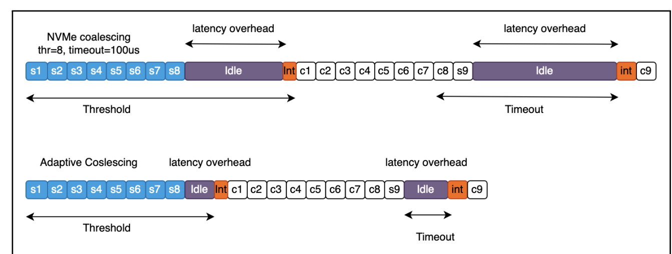

Figure 2: The problem of using original NVMe coalescing which can introduce high delay and timeout increasing latency.

Figure 2 shows the actual problem that we have under the current approach. Problem can be described with following points:

- NVMe coalescing uses static value for threshold and timeout which can introduce worse case latency of timeout for a IO request which is completed in very small time and is very latency sensitive.
- Although Adaptive coalescing solves the above mentioned problem upto a extent but c1 which is very latency sensitive IO request need to wait until other X request are completed to meet the threshold value which might not be latency sensitive.

In the context of Linux systems utilizing NVMe devices, a notable gap exists regarding balancing interrupt reduction with latency management for latency-sensitive tasks.

Interrupt coalescing, a common technique for minimizing interrupt frequency, can introduce latency for I/O requests. This latency stems from waiting to accumulate a batch before processing, potentially impacting the responsiveness of latency-sensitive applications.

The gap arises as current approaches overlook latency sensitivity at the application layer during interrupt coalescing. These methods prioritize reducing interrupt overhead without

considering the distinct latency requirements of different I/O requests.

Closing this gap requires a deeper understanding of how interrupt coalescing affects diverse applications and workloads. By integrating latency sensitivity at the application layer, it's possible to prioritize and manage I/O requests based on their latency needs. This approach aims to minimize the impact of interrupt storms on latency-sensitive tasks while leveraging the efficiency gains of interrupt coalescing.

In summary, addressing this research gap highlights the importance of a nuanced strategy in NVMe-based storage systems. This involves incorporating latency sensitivity into the application layer to optimize performance effectively.

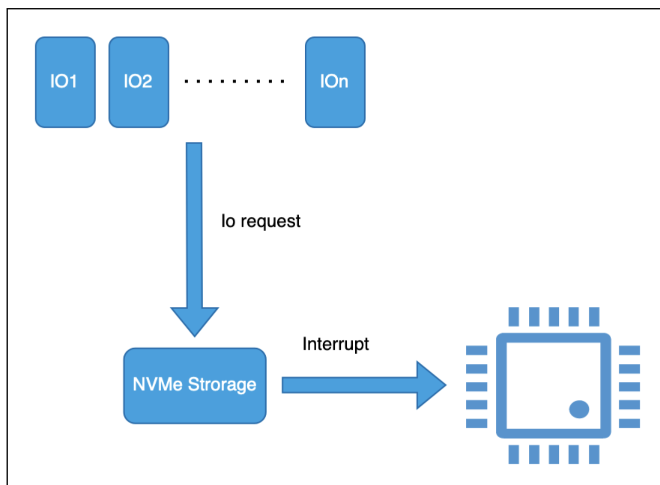

Figure 3: Interrupt coalescing treats all IO request as same, Here IOx can be latency sensitive but coalescing treats this request similar to other which would decrease the performance of application

2. Literature Review

The emergence of high-performance storage technologies, exemplified by Non-Volatile Memory Express (NVMe) SSDs, has prompted innovative approaches to optimize their utilization. The paper on NVMeDirect by Kim et al. [13] introduces a user-level I/O framework designed to enhance the performance of NVMe SSDs. By enabling direct access without kernel intervention, NVMeDirect effectively addresses the limitations of traditional kernel-based I/O stacks. This user-level framework not only facilitates improved performance, surpassing kernel-based I/O in microbenchmarks and applications such as Redis, but also introduces flexibility for user applications to define their I/O policies, including I/O completion methods, caching, and I/O scheduling. The study provides a significant contribution to the evolving landscape of storage system optimization, emphasizing the importance of user-level frameworks in unlocking the full potential of modern storage devices.

The integration of Berkeley Packet Filter (BPF) support in Linux has streamlined the incorporation of kernel extensions. Recent studies have explored the use of BPF to offload sections of server applications, showcasing enhanced performance and efficiency. However, a key question remains: how to selectively identify components for offloading within an application. The paper "Automatic Kernel Offload Using BPF" [14] addresses this by highlighting the drawbacks of blind offloading and advocating for an automated decision-making process. The proposed solution involves a compiler that analyzes application code, generating separate kernel offload and userspace program executables. Challenges associated with building this compiler are discussed, offering a feasible approach to automate decision-making and optimize BPF-based kernel offloading performance.

Zhong et al.'s paper on XRP [15] addresses the growing challenge posed by the kernel storage stack as a bottleneck for high-performance storage devices. The authors introduce XRP, a solution leveraging BPF to offload storage functions to the Linux kernel. However, this approaches has various security issues and limited usability. This only works well when data is stored in the predefined structures like B-trees.

Smotherman's historical account [16] traces the evolution of interrupts and their application across diverse computer systems, commencing with UNIVAC in 1951. As network bandwidth and storage device IO throughput have steadily increased, the frequency of interrupts, and consequently, the CPU overhead required to manage them, has followed suit since the inception of the interrupt model. Despite advancements in processor speeds and the proliferation of cores, the persistent aim has been to minimize the overall CPU overhead associated with interrupt handling. A notable solution to address this challenge is interrupt coalescing, a strategy that has proven highly effective in hardware controllers. Numerous patents and scholarly papers have delved into the implementation of interrupt coalescing for both network and storage hardware controllers, demonstrating its successful deployment and ongoing relevance in contemporary computing environments.

In the examination of various interrupt handling schemes, Salah et al. [17] conducted a comprehensive analysis encompassing polling, regular interrupts, interrupt coalescing, as well as disabling and enabling interrupts. Their findings underscore the absence of a universally superior scheme across all traffic conditions. This conclusion accentuates the pressing need for adaptive mechanisms capable of dynamically adjusting to prevailing interrupt arrival rates and other workload parameters. Building on this premise, Salah [18] performed an analytical and simulation study specifically comparing the benefits of time-based versus number-of-packets-based interrupt coalescing in the realm of networking. Furthermore, Salah and Qahtan [19] contributed to this discourse by implementing and evaluating a distinct hybrid interrupt handling scheme tailored for Gigabit Network Interface Controllers (NICs) within the Linux kernel version 2.6.15. Their hybrid approach seamlessly toggles between interrupt disabling-enabling (DE) and polling, demonstrating a commitment to addressing the intricate challenges associated with interrupt management.

The paper presented by Amy et al. [7] sheds light on a critical limitation associated with the existing NVMe interrupt coalescing API, highlighting its impracticality for effective coalescing. In response to this challenge, the authors propose an adaptive coalescing strategy tailored specifically for NVMe.

A key insight from their research is the assertion that software directives emerge as the optimal method for a storage device to generate interrupts. The proposed solution, referred to as Cinterrupts, combines the functionalities of Urgent, Barrier, and an adaptive burst-detection strategy. By doing so, Cinterrupts excels in generating interrupts precisely when the workload necessitates them, thereby enhancing overall performance even in dynamic environments. This novel approach empowers the software stack to fully leverage the capabilities of both existing and forthcoming low-latency storage devices, thereby addressing a crucial aspect of interrupt management in the context of NVMe technology. However, the algorithmic design proposed in the paper does not fully solve the problem which we try to solve in this research.

3. Methodology

The core insight of this research is that device-level heuristics for coalescing interrupts are limited because of a semantic gap between the device, which sees a stream of requests, and the requester, which knows which requests require interrupts to unblock the application. To bridge this gap, the research proposes that the application issuing the I/O request should inform the device when it wishes to be interrupted.

The proposed solution, enhances adaptive coalescing with two Flags:Urgent and Barrier, which software passes to the device.

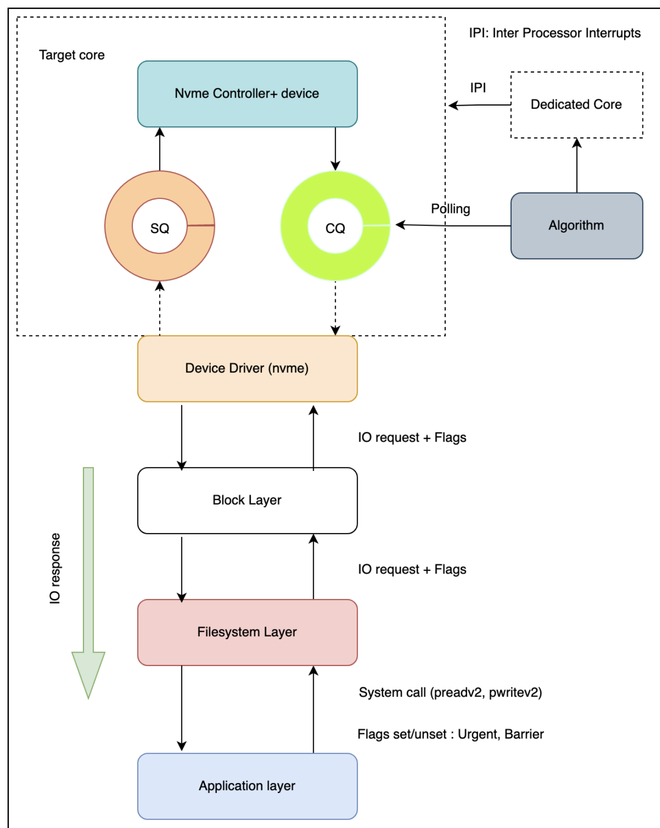

Figure 4: Flags with Application layer semantics are propagated to device driver , where coalescing algorithm acts on it and generates a IPI: Inter Processor Interrupts for coalesced IO requests

Urgent: Urgent Flag is used to request an interrupt for a single request, allowing the device to generate an immediate interrupt for any request annotated with Urgent. This is particularly useful for calibrating interrupts for latency-sensitive read requests, reducing latency without generating unnecessary interrupts that could impact throughput.

Barrier: Barrier Flag is used to calibrate interrupts for batches of requests. It marks the end of a batch and instructs the device to generate an interrupt as soon as all preceding requests have finished. Unlike Urgent interrupts, Barrier interrupts may have to wait if requests are completed out of order. Barrier minimizes the interrupt rate, which is beneficial for CPU utilization, while ensuring that the device generates enough interrupts so that the application is not blocked.

Along with the introduction of Urgent and Barrier flag we have implemented following algorithm to utilize the hybrid of adaptive algorithm and addition of user layer semantics to create bearable interrupts , throughput with low latency with currently available static and adaptive interrupt methodology.

3.1 Algorithm

The proposed algorithm 1 represents an enhanced interrupt coalescing mechanism designed to optimize the management of interrupts in a dynamic computing environment. The key motivation behind this algorithm is to efficiently handle interrupt requests, balancing the need for timely completion delivery and minimizing CPU overhead. The algorithm operates based on the parameters Δ and *thr*, denoting the time interval and the coalescing threshold, respectively.

At its core, the algorithm 1 continuously monitors the system for incoming interrupt completions. It dynamically adjusts the timeout period based on the calculated delta, ensuring adaptability to the current interrupt arrival rate and workload conditions. Upon the arrival of a completion, the algorithm evaluates its type, distinguishing between Urgent, Barrier, and regular completions.

In the case of Urgent completions, the algorithm intelligently decides whether to process only urgent requests (if out-of-order processing is enabled) or handle all requests. Barrier completions trigger immediate processing, while regular completions contribute to a coalescing counter. The algorithm 1 checks if the coalescing threshold is reached, firing an interrupt and resetting the coalescing counter accordingly.

The algorithm 1 introduces a quiescent period concept, where if coalesced requests are pending, it initiates an interrupt to address accumulated completions. This mechanism optimizes interrupt handling, preventing unnecessary delays and efficiently utilizing system resources.

A noteworthy feature of the algorithm is the calculation of Δ using the COMPUTEΔ function. The delta computation is dynamically adjusted based on the difference between the specified threshold and the actual number of coalesced requests. As demonstrated in Figure 5, the relationship between Δ and **a** a.k.a correction factor is that Δ is influenced by the exponential growth determined by correction factor in

Algorithm 1 Improved Coalescing Algorithm

```
Parameters: Δ, thr
coalesced ← 0
timeout ← now + COMPUTEΔ(thr, coalesced)
while true do
    while now < timeout do
        while new completion arrival do
            timeout ← now + COMPUTEΔ(thr, coalesced)
            if completion type == Urgent then
                if out-of-order processing is enabled then
                    FIREURGENTIRQ()
                else
                    FIREIRQANDRESETCOALESCED()
                end if
            else if completion type == Barrier then
                FIREIRQANDRESETCOALESCED()
            else
                coalesced ← coalesced + 1
                if coalesced ≥ thr then
                    FIREIRQANDRESETCOALESCED()
                end if
            end if
        end while
        if coalesced > 0 then
            FIREIRQANDRESETCOALESCED()
        end if
        coalesced ← 0
        timeout ← now + COMPUTEΔ(thr, coalesced)
    end while
end while
function COMPUTEΔ(thr, coalesced)
    a ← 0.1                        ▷ Adjust the value of 'a' as needed
    return e^{a(thr−coalesced)} − 1
end function
```

the context of the given algorithm. Larger values of correction factor will result in more rapid changes in Δ concerning changes in Available Coalescing Space, while smaller values of correction factor will produce more gradual variations. Adjusting correction factor allows you to control the sensitivity of Δ to changes in Available Coalescing Space.

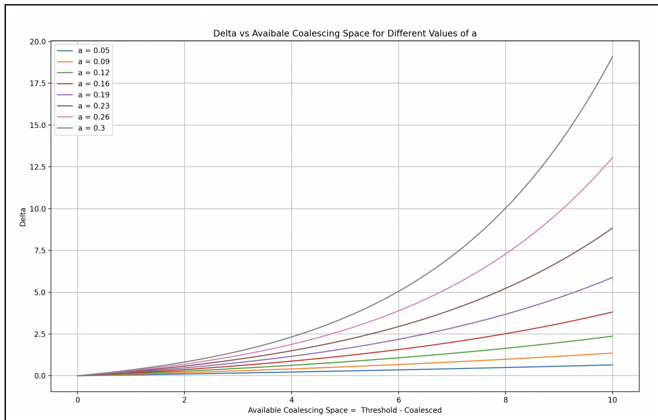

Figure 5: The parameter a determines the rate at which the exponential term grows. Higher values of a lead to a steeper exponential growth, while lower values result in a more gradual increase.

This adaptive approach ensures that the timeout period aligns with the current workload, striking a balance between interrupt responsiveness and CPU utilization. This algorithm addresses the challenges posed by interrupt handling by introducing adaptability through dynamic delta computation. Its key contributions lie in the effective coalescing of interrupts, minimizing CPU overhead, and optimizing completion delivery in a diverse and dynamic computing environment. The implementation for this research includes Application layer and kernel layer modifications.

3.2 Kernel Modifications

The kernel exposes system calls to enable user layer applications to perform actions on peripherals. For instance, disk read/write operations cannot be directly executed by the application layer; instead, the user layer application utilizes system calls exposed by the kernel for these operations. In the context of our research, we have made modifications to the preadv2 and pwritev2 system calls. We extended these system calls to allow the application layer to pass a one-bit information through flags associated with these system calls. This one-bit information is then propagated to the file system layer, block layer, and eventually to the device driver (NVMe). The NVMe driver attaches this information to NVMe command flags, and the requests are then placed into the submission queue. Once these I/O requests are processed, they reside in the completion queue.

The algorithm described above needs to be implemented in the NVMe firmware itself to perform coalescing and generate interrupts accordingly. However, the firmware changes are not within the scope of this research. Instead, we have emulated the interrupt generation part using a completion queue polling strategy in the device driver. A kernel thread is created to poll the completion queue for requests, which are then coalesced based on the algorithm discussed in this research paper. When the timeout/threshold value is reached or the interrupt generation condition is met, the NVMe driver generates an Inter-Processer Interrupt (IPI). Since this research utilizes a polling strategy, a dedicated CPU is used for polling. This overhead could be avoided if the algorithm were implemented directly in the firmware.

3.3 Application Layer Modifications

Macro benchmarking for this research utilized LevelDB, a fast key-value storage library. Minor modifications were made to the LevelDB disk read/write interface to accommodate the research requirements. An option named enable_urgent was introduced in LevelDB, which can be specified when creating a database (DB) instance. When this option is enabled, all foreground get/put operations are marked as urgent when calling the preadv2 and pwritev2 system calls. In contrast, background operations such as compaction, logging, checkpointing, and backup/restoration are marked as non-urgent I/O requests.

This approach allows for the prioritization of urgent operations, ensuring that they receive timely attention while non-urgent operations can proceed in the background without impacting critical tasks.

3.4 Experimental Scenario

The experiment was conducted on a virtualized environment running the Ubuntu 14.04 operating system. Amazon Web Services (AWS) cloud services were leveraged as the Infrastructure as a Service (IaaS) provider. The experiment utilized the c5d.4xlarge EC2 instance type, which provides a high-performance computing environment. This instance type is equipped with 16 virtual CPUs (vCPUs), 32 GB of RAM, and a high-speed 450 GB NVMe SSD storage device. To ensure compatibility with the experimental setup, a modified kernel was used. This modified kernel included changes in the nvme driver, filesystem layer and block layer. The performance of the system was evaluated using modified system calls preadv2 and pwritev2, The benchmarking process involved executing these modified system calls under various conditions to assess the Nvme performance.

4. Benchmarking

Micro Bench-marking of pread64 and pwrite64 sycalls with variable workload , synchronous and asynchronous IO operation has been carried out.

Figure 6 shows that the default configuration i.e interrupt per IO request has very low latency but the interrupt rate is high which can lead to interrupt storm and high cpu utilization leading to low IOPS. Current nvme uses static coalescing strategy with timeout value min to 20us and max 100us, we evaluate adaptive strategy against the following configurations: no coalescing (default), nvme100 (timeout of 100us, the minimum standard NVMe timeout), nvme20 (theoretical timeout of 20us), and nvme6 (theoretical timeout of 6us), under these configurations the latency is high in comparison to default strategy this is because every I/O request are treated as same without considering the urgency of these request to application layer, since these strategy uses static timeout value Interrupt for I/O request has to wait until timeout is reached or threshold value is reached , introducing high latency.Adaptive strategy which we have utilized in this research paper shows the latency metrics similar to the default i.e no coalescing with reduced interrupt as compared to default strategy, current available static coalescing algorithm with 100us timeout and 10us timeout.This benchmark was carried out by generating 4kb uniform read/write by 2 threads with preadv2 and pwritev2 system calls present in Linux systems.

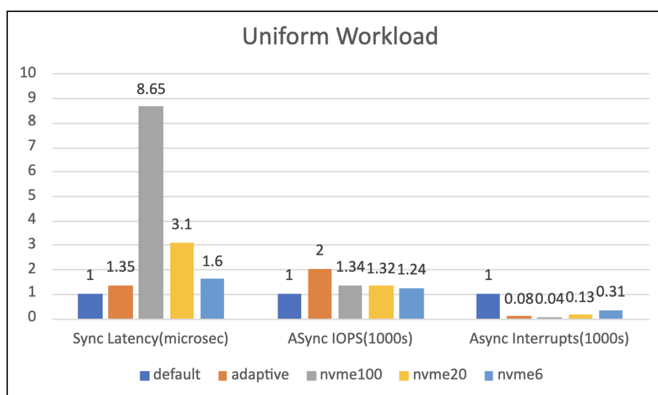

Figure 6: The adaptive strategy exhibits superior performance for uniform workload, irrespective of the configuration of NVMe coalescing. This superiority is evident in various aspects: (a) reduced latency for synchronous read requests, (b) enhanced throughput for asynchronous read workloads characterized by high iodepth, and (c) a lower interrupt rate for asynchronous workloads. The labels in the results indicate the performance improvement relative to the default configuration.

Figure 7 shows that the adaptive coalescing strategy doesn't work well in varying workload scenario, this is because varying workload can sometime generate burst of I/O request and other time generate very low number of I/O request on the system, under such workload adaptive strategy has increased latency,this shows that any coalescing strategy used in kernel without analyzing the application layer information will have some problems in variable workload scenario. This benchmark was carried out by generating burst of I/O workloads from a thread and delayed workload from another thread.

Figure 8 illustrates how implementing cinterrupts can enhance IOPS, lower latency, and decrease the number of interrupts. This improvement is achieved through the

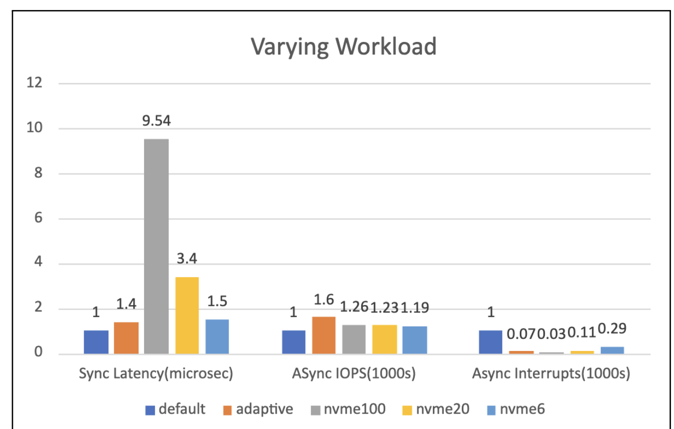

Figure 7: The adaptive strategy exhibits superior performance for varying workload as well. But (a) latency has increased, (b) async IOPS is decreased and (c) number of interrupts decreased as compared to uniform workload.

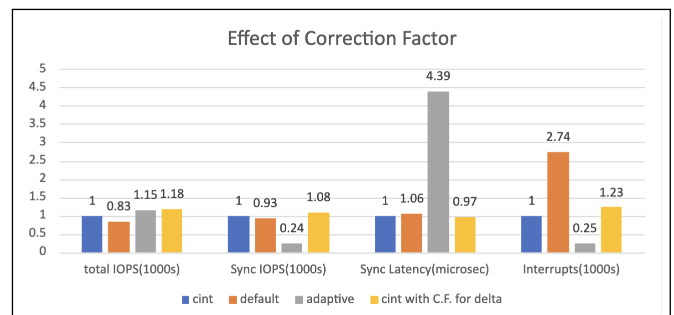

Figure 8: Effect of cint and correction factor: The introduction to calibrated interrupts based on user layer semantics(with the use of urgent and barrier flags) (a)(b) improves IOPS, (c) decreases latency and (d) decreases the number of interrupts. Further, the introduction of a correction factor enhances (a) (b) IOPS even further and (c) reduces latency more effectively. However, (d) it slightly increases the number of interrupts.

utilization of user layer semantics flags, namely Urgent and Barrier. However, with variable workloads, employing a static delta may negatively affect IOPS. Therefore, by incorporating a delta with a correction factor, as outlined in Algorithm 1, IOPS is increased and latency is significantly reduced. Nonetheless, it's worth noting that this approach might lead to a slight increase in the number of interrupts compared to using a static delta.

5. Conclusion

In conclusion, our research highlights the substantial limitations of the current NVMe interrupt coalescing approach in practical coalescing strategies. Through our investigation, we have developed an adaptive coalescing approach for NVMe that effectively mitigates these limitations.

Incorporating user layer semantics to define the latency sensitivity of IO requests represents a significant advancement in reducing system latency and interrupts. However, our research has revealed that this approach, when combined with traditional NVMe coalescing algorithms, is not effective in scenarios with variable workloads or systems where the majority of IO requests are latency-sensitive.

By introducing a timeout correction factor and dynamic timeout calculation algorithm, we have demonstrated the effectiveness of combining user layer information with our adaptive interrupt coalescing strategy. Our findings indicate substantial benefits over previous approaches, particularly in environments with variable workloads.

In summary, our research contributes a novel approach to NVMe interrupt coalescing that significantly improves system performance and efficiency under diverse workload conditions.

References

[1] John Uffenbeck and 8088 Family. *The 80x86 Family: Design, Programming, and Interfacing.* Prentice Hall PTR, USA, 2nd edition, 1997.

[2] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *SYSTOR '13 Proceedings of the 6th International Systems and Storage Conference. Association for Computing Machinery, 22. (Systor).*, page 1, United States, 2013. Association for Computing Machinery.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Symposium on Networked Systems Design and Implementation*, 2012.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: a protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 49–65, USA, 2014. USENIX Association.

[5] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. pages 161–176, 06 2017.

[6] Livio Soares and Michael Stumm. Flexsc: flexible system call scheduling with exception-less system calls. OSDI'10, page 33–46, USA, 2010. USENIX Association.

[7] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. *ACM Trans. Storage*, 18(1), mar 2022.

[8] Microsoft Corporation. Microsoft documentation: Optimize performance on the lsv2-series virtual machines, 2019. Accessed: May, 2021.

[9] K.R. Fall and W.R. Stevens. *TCP/IP Illustrated: The Protocols, Volume 1.* Addison-Wesley Professional Computing Series. Pearson Education, 2011.

[10] Fernando Gont. Survey of security hardening methods for transmission control protocol (tcp) implementations. Technical report, Internet Engineering Task Force, March 2012.

[11] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), dec 2016.

[12] Intel. ntel-ssd-dc-p3700-series-400gb-1-2-height-pcie-3-0-20nm-mlc. OSDI'10. Intel Corporatiom, 2021.

[13] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds. 2016.

[14] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. Automatic kernel offload using bpf. *Workshop on Hot Topics in Operating Systems (HOTOS '23)*, 2023.

[15] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. Xrp: In-kernel storage functions with ebpf. 2022.

[16] M. Smotherman. Interrupts, 2008.

[17] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Comput. Commun.*, 30(17):3425–3441, 2007.

[18] K. Salah. To coalesce or not to coalesce. *Intl. J. of Elec. and Comm.*, pages 215–225, 2007.

[19] K. Salah and A. Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Comput. Commun.*, 32(1):179–188, 2009.