applied sciences

MDPI

Article

uDMA: An Efficient User-Level DMA for NVMe SSDs †

Jinbin Zhu [1,2], Liang Wang [1,2,*], Limin Xiao [1,2,*] and Guangjun Qin [3]

[1] State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China
[2] School of Computer Science and Engineering, Beihang University, Beijing 100191, China
[3] Smart City College, Beijing Union University, Beijing 100101, China
* Correspondence: lwang20@buaa.edu.cn (L.W.); xiaolm@buaa.edu.cn (L.X.)
† This paper is an extended version of our paper published in "UPM-DMA: An Efficient Userspace DMA-Pinned Memory Management Strategy for NVMe SSDs" published in the 21st International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2021), Xiamen, China, 3–5 December 2021.

Abstract: The Non-Volatile Memory Express (NVMe) SSD provides high I/O performance for current computer systems, and direct memory access (DMA) is the critical enabling mechanism for direct I/O. However, the lengthy I/O stack becomes a new bottleneck that degrades the potential of NVMe SSD. This paper reveals that existing user-level DMA introduces additional overhead for pinning memory used by DMA from the user space. Moreover, it cannot adapt to I/O requests of different data sizes. This paper proposes an efficient and dynamically adaptive user-level DMA (uDMA) mechanism that can adapt to I/O requests for different data sizes and lighten the I/O software stack by amortizing per-request latency. The critical component of uDMA is the pinned memory pool, which avoids frequently pinning new memory blocks by reusing allocated and pinned memory blocks. In addition, it effectively connects the discrete pinned memory blocks by the scatter/gather lists, improving the utilization of the pinned memory pool. Compared with the latest user-level DMA method, uDMA has an improvement of at least 17% under various data sizes.

Keywords: NVMe SSD; user-level DMA; scatter/gather lists; pinned memory pool

Citation: Zhu, J.; Wang, L.; Xiao, L.; Qin, G. uDMA: An Efficient User-Level DMA for NVMe SSDs. *Appl. Sci.* **2023**, *13*, 960. https://doi.org/10.3390/app13020960

Academic Editor: Paolino Di Felice

Received: 30 November 2022
Revised: 30 December 2022
Accepted: 7 January 2023
Published: 10 January 2023

1. Introduction

Today's popular cloud computing workloads (e.g., Hadoop [1], RocksDB [2]) exert increasingly intense pressure on the I/O systems, posing higher performance requirements for new storage devices [3]. The Non-Volatile Memory Express (NVMe) [4] SSD is emerging and has become a widely used storage solution for modern computer systems [5,6]. For example, the Intel Optane SSD series provide the capability of reading and writing data up to 2.5 GB/s and 1.2 GB/s [7]. The NVMe SSD has attracted many researchers to design SSD-oriented high-performance storage systems [8–10]. Many cloud providers (Alibaba [11], Amazon, et al. [12]) have configured NVMe SSDs to build high-performance storage systems.

However, although an NVMe SSD dramatically improves the hardware performance of storage devices, the lengthy I/O software stack in an operating system (OS) degrades the potential of the NVMe SSD. The latest research from Intel shows that the Linux kernel takes up too much execution time on the I/O stack in NVMe SSD-oriented storage system [13–15]. The main reason is that context switching and interrupting waste much time. For example, context switching must be between the user and the kernel modes when performing DMA operations. Frequent processing of I/O requests can cause frequent switching, resulting in a high switching overhead.

Recently, many studies [13,14,16–18] of NVMe SSDs on the I/O software stack have tended to access NVMe SSDs directly from the user space. User-level methods move a partial kernel I/O stack in the OS into the user space that can eliminate context switches between the Linux kernel and the user space. For example, SPDK [13] is a high-performance user-level storage tool library developed by Intel. It utilizes polling mode, lock-free, and

user-level direct memory access (DMA) to provide highly parallel access to NVMe SSDs from user-space applications. User-level DMA is a lightweight data transfer mode that can transfer data directly from or to NVMe SSDs without involving the CPU. However, this paper revealed that the user-level methods did not perform well in data-intensive workloads. Taking SPDK as an example, the user-level DMA mechanism implemented in SPDK brings some additional overhead. Specifically, SPDK uses the spd_dma_malloc() [19] function to allocate and pin memory to transfer data. It needs to perform pin memory operations, which can incur a considerable overhead under data-intensive workloads.

In this paper, we propose an efficient and dynamically adaptive user-level DMA (UDMA) mechanism for different I/O sizes for NVMe SSDs, which lightens the I/O software stack by amortizing per-request latency. The core of uDMA is an efficient memory pool, which implements a pinned memory pool for I/O requests without frequently allocating, pinning, and freeing new memory. In addition, uDMA designs a DMA memory management strategy based on scatter/gather lists, which can integrate discrete memory blocks for data transmission and improve the efficiency of memory use. Our preliminary work was presented in [16]. This paper extends the preliminary work by adding a pinned memory block management strategy based on the scatter/gather list. The new design can connect discrete pinned memory blocks to form a large, pinned memory region, improving the efficiency of pinned memory usage.

This paper makes the following contributions:

- We find that the original user-level DMA mechanism, which aims to directly access an NVMe SSD from the user space, does not perform well in data-intensive cases. It needs to perform pin memory operations, which can incur a huge overhead under data-intensive workloads.
- We design uDMA, a new user-level DMA mechanism that reduces the I/O software stack's overhead. First, it uses a pinned memory pool to reduce the initialization overhead of DMA memory. Second, the proposed pinned memory pool improves memory efficiency by scatter/gather lists.
- We implement uDMA as a memory library called uDMA libs and integrate it into the SPDK framework. The experimental results show that uDMA improves the I/O performance of NVMe SSDs by at least 17%.

The remainder of this paper is organized as follows. In Section 2, related works are discussed. In Section 3, the background and motivation are presented. In Section 4, the proposed uDMA is introduced. In Section 5, the experimental setup and evaluations are presented. Finally, this work is concluded in Section 6.

2. Related Work

With the continuous improvement of the reading and writing performance of storage devices (e.g., NVMe SSDs), the I/O software stack in the OS has become unsuitable, degrading the potential of fast storage devices. According to the latest research results of Intel, the existing Linux kernel takes up too much execution time on the I/O stack in NVMe SSD-oriented storage systems [13–15]. The main problem in the OS's I/O software stack is to reduce the context switching and data-copying overhead between the user space and the kernel, the interrupt processing overhead of I/O requests, the competition overhead of shared resources in the kernel I/O stack, etc.

Several optimization methods have been proposed to reduce the time cost of the I/O software stack [20–24]. For example, Lee et al. [22] proposed a new I/O scheduler that allowed garbage collection processing to be preempted while I/O requests were pending. With the support of NVMe SSD interfaces, Kim et al. [24] streamlined the I/O path to exploit the performance characteristics of NVMe SSDs. However, traditional methods to solve these problems mainly include polling, merging I/O, parameter-aware I/O scheduler, etc. These performance improvements need to be greater for an NVMe SSD, a fast storage device [13].

To fully utilize the performance of NVMe SSDs, the user-mode I/O frameworks have been proposed in recent years, which significantly reduce the time overhead of the I/O

software stack by avoiding context switching. Researchers in [13,14,17,18] designed SPDK, NVMeDirect, and UNVMe, all of which were based on user-mode I/O frameworks, where SPDK was released and continuously maintained by Intel. For example, SPDK eliminates context switches and interrupts handling overhead by moving kernel drivers into the user space. NVMeDirect combines SPDK with the traditional I/O stack. Compared with the kernel-based NVMe driver, SPDK can improve IOPS by 6× to 10×, and NVMeDirect outperforms others by up to 30% on microbenchmarks and up to 15% on accurate benchmarks. Although these methods can significantly improve I/O performance, they also introduce new problems. For example, the virtual and physical addresses' mapping relationship may be modified when the user directly accesses the fast storage device. Therefore, pinning the memory during DMA operation is used to keep the mapping relationship from the virtual address to the physical address unchanged during DMA operation.

This pinned memory operation has a non-negligible overhead. In the face of I/O-intensive applications, frequent pinned memory will seriously degrade the execution efficiency of I/O requests. Currently, the research on DMA buffer operation mainly focuses on the network. For example, there are many pieces of research on RDMA (remote direct memory access) buffers [25], but there are few studies on DMA buffers when users directly access fast storage devices. Although researchers in [26] studied the issue of DMA caching, they mainly considered it from a security perspective.

Different from the existing methods, this paper proposes an efficient and alternative UDMA mechanism. It is dynamically adaptive for different I/O sizes for NVMe SSDs and lightens the I/O software stack by amortizing per-request latency. uDMA implements a pinned memory pool for I/O requests without frequently allocating, pinning, and freeing new memory. In addition, uDMA designs a DMA memory management strategy that integrates discrete memory blocks by scatter/gather lists to improve the efficiency of memory use.

3. Background and Motivation

In this section, we first introduce the background of DMA-based data transfer and user-level DMA. Then, we present our motivation to investigate uDMA to reduce the overhead caused by pinning memory for user-level DMA.

3.1. Background

DMA is an excellent feature of modern computer systems. With the help of DMA, applications in OS can transfer data between specific PCIe-based devices and the main memory independently, without involving the CPU. Without DMA, when a process performs I/O operations, it typically fully occupies the CPU for the entire life cycle of the I/O requests, making the CPU unable to perform other works. With DMA, the CPU only needs to initialize the data transfer, such as the data's direction, size, and location. CPU can then do other work. This feature is critical to improving the performance of asynchronous I/O requests because the CPU does not need to wait for relatively slow I/O data transfers.

Compared to the DMA in kernel space, user-level DMA is a more lightweight data transfer mode. It allows users to access data for DMA transfer or control DMA transfer from the user space. Specifically, NVMe SSDs can be accessed directly through UIO (user space I/O) or VFIO (virtual function I/O) at the user level. Based on UIO and VFIO, users can implement user-level DMA, assigning a hardware device to a process that allows the process to operate and read/write the device. However, UIO and VFIO must solve a critical problem: ensuring that physical memory is in place during the DMA process. Currently, the mainstream accepted method is manually pinning the physical memory pages. Pinning the memory pages makes these pages unable to be modified (e.g., page swapping). Pinning memory pages can cause a certain degree of I/O performance loss, especially for I/O-intensive applications.

3.2. Motivation

To assess the effectiveness of the existing user-level DMA mechanism, we evaluated the state-of-the-art SPDK and the traditional memory copy method in Linux. We evaluated the overheads associated with memory page pinning and copying under various data sizes. First, we used SPDK's *spdk_dma_malloc* each time to allocate and pin memory pages for different data sizes. Then, the pinned memory pages were released immediately. We performed this operation ten more times and calculated the average time cost. After that, we copied the data to the pinned memory for the same testing data sizes and calculated the average overhead caused by data copying. Figure 1 shows the statistical results.

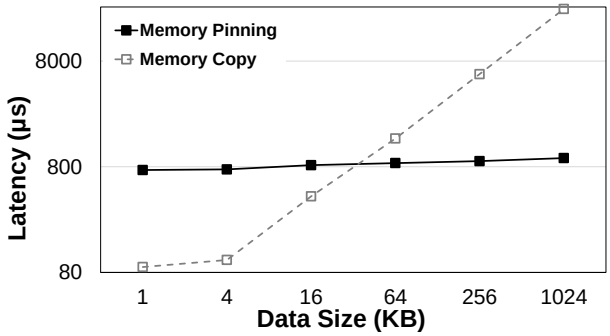

Figure 1. Time consumption of pin memory and memory copy.

We identified two major shortcomings of the latest user-level DMA mechanism: (1) pinning memory page operation has a specific time overhead, and (2) a single user-level DMA method cannot be suitable for all I/O requests with various data sizes.

(1) Pinning memory page operation has a specific time overhead. Figure 1 shows the cost of pinning memory pages. First, pinning one memory page (e.g., 1 KB or 4 KB) takes about 800 µs. Second, with the increase in memory page size, the time cost of pinning memory pages increases gradually. In addition, as the number of pinning memory pages increases, so does the time overhead. Especially for I/O-intensive applications, frequently pinning memory pages can bring a huge time overhead, increasing the processing time of the whole I/O software stack in the OS. This can seriously degrade the potential of NVMe SSDs.

(2) A single user-level DMA method cannot be suitable for all I/O requests with various data sizes. As shown in Figure 1, when the data size is less than 60 KB, the overhead of pinning memory pages is more significant than the memory copy. However, as the data size increases, the time taken to copy memory is gradually greater than the cost of pinning memory. In addition, the time overhead of the memory copy will continue to increase. Therefore, there is a tradeoff between pinning memory pages and memory copies.

We make the following conclusions from the observation. First, for small data requests, copying data from pageable to pinned memory is shorter than pinning memory. Furthermore, because of the small quantity of data, only a few memory blocks need to be pinned to meet the requirement. It means we can pin a small piece of memory pages during the initialization phase and keep it pinned throughout the application's life cycle. Second, we can build a pinned memory pool for medium data requests to avoid frequent allocating, pinning, and freeing memory blocks. When a new I/O request arrives, if the available memory in the pinned memory pool meets the needs of the I/O request, we can directly fetch the corresponding memory block and assign it to the I/O request without allocating and pinning the new memory block. Finally, for the I/O requests of big data, the time consumption of pinning memory pages is less than that of the memory copy. However, pinning a large piece of memory blocks for a long time can lead to inefficient memory use and bring the lack of memory to other system applications. Therefore, we can only dynamically allocate and pin memory blocks for these requests.

4. Design of uDMA

uDMA is an efficient and dynamically adaptive user-level DMA mechanism for different I/O sizes for NVMe SSDs. It can lighten the I/O software stack by amortizing per-request latency. Different from the latest user-level DMA, uDMA integrates three different pinned memory management strategies. For small data requests, it provides a statically pinned memory management method. It builds a pinned memory pool for medium data requests to alleviate frequently allocating and pinning memory pages. It adopts the strategy of dynamic allocating and pinning memory pages for big data requests.

4.1. Overview of uDMA

Figure 2 shows the framework of uDMA. We adopted the statically pinning memory strategy for small I/O requests equal to or less than 4 KB. Thus, we allocate and pin three memory blocks during the library initialization process. Users can set the size of memory blocks according to their experience. These memory blocks remain pinned throughout the life cycle of the application. Because of NVMe SSDs' fast reading and slow writing characteristics, we use one piece of memory for reading requests and the remaining two blocks for writing requests. Our design is because small data I/O requests do not need to take up too much memory. If we maintain a pinned memory state throughout the application's life cycle, we do not waste memory, and we only need to spend one-time pinning memory.

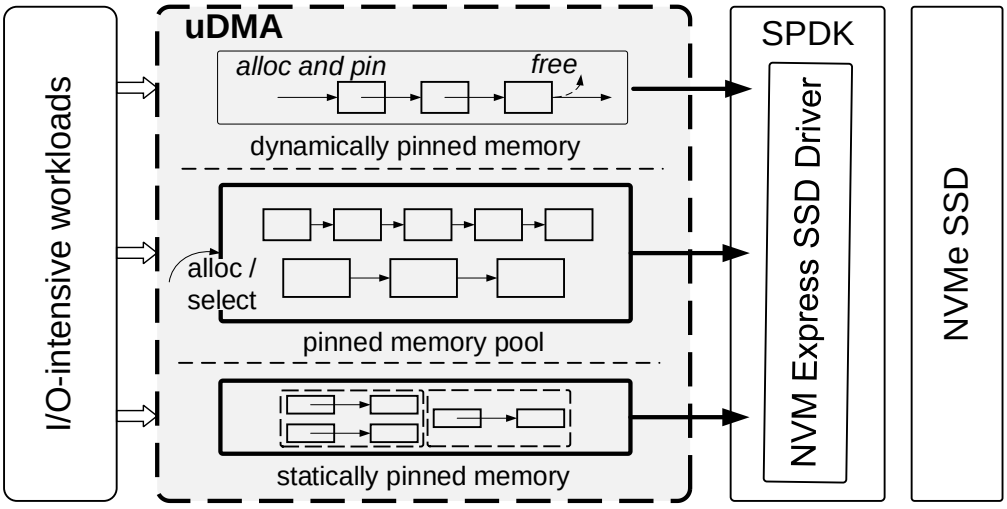

Figure 2. The framework of uDMA.

We designed a pinned memory pool for medium data I/O requests and proposed allocation and release algorithms. Unlike small I/O requests and big I/O requests, the copy and pinning memory costs for medium data are relatively high. The typical way to process these is to pin the memory of the corresponding size for user-level DMA operation for each I/O request. In the face of I/O-intensive applications, the pinned memory area cannot be reused, and then the processed I/O requests need to be pinned again, which introduces a lot of pinned memory overhead.

For a big data's I/O request, which is greater than or equal to 4 MB, we adopted the strategy of dynamically pinned memory. This strategy dynamically allocates and pins memory blocks when the application reads from and writes to NVMe SSDs through user-level DMA. We unpin and release the pinned memory block immediately after the user-level DMA operation. We cannot use the static pinned memory strategy for I/O requests for big data because these requests take up more memory. If we occupy memory for a long time, it can have a destructive impact on other processes. Another reason is that the pinned memory time is less than the big data's copy time.

4.2. Statically Pinned Memory Management

The statically pinned memory management is shown in Figure 3. It contains three memory block lists, namely (a), (b), and (c). Memory blocks (a) and (b) serve the writing requests of the NVMe SSD. We designed two memory blocks to serve writing requests because of the imbalance between the NVMe SSD's reading and writing. Because the writing speed is relatively slow, the data in the memory block take a long time to write to the NVMe SSD. As shown in the figure, when (a) is full, it is out of service, and the data in it are written to the NVMe SSD in batch. At the same time, (b) begins to serve writing requests. When (b) is full, (a) becomes free. The (c) list illustrated in Figure 3 is designed for reading requests. The principle is based on the feature of reading fast and writing slowly of the NVMe SSD. Assigning one list of memory blocks for reading requests can meet reading requirements while saving memory resources.

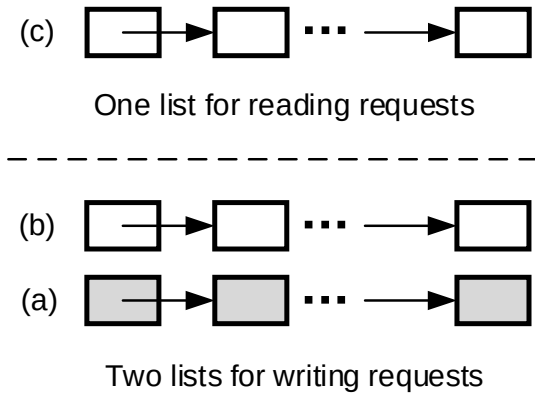

Figure 3. Two pairs of statically pinned memory block lists.

This design has three points: (1) It can reduce the data transfer time, and we described the reasons in detail in the motivation section. (2) It takes up less memory space because the design is oriented toward small data requests. Thus, we only need to fix a few memory blocks to meet the requirements. (3) It fully considers the characteristics of fast reading and slow writing of NVMe SSDs, prepares enough memory space for write requests, and improves the processing efficiency of NVMe SSDs' write requests.

4.3. Pinned Memory Pool

Figure 4 shows the designed pinned memory pool for medium data I/O requests. For a newly arrived I/O request, we first try to find out if there is a memory block that meets the requirements in the pinned memory pool. If it exists, we can directly fetch the corresponding memory block and assign it to the request. We allocate and pin the new memory block if it does not exist. When the I/O request is completed, the newly pinned memory block is saved in the memory pool for use by subsequent I/O requests. In addition, we designed a pinned memory block management strategy based on the scatter/gather list. Traditional DMA memory must be contiguous, resulting in low memory utilization. For example, even if there are enough pinned memory blocks in the memory pool, they cannot be utilized if they are discontiguous. uDMA uses the scatter/gather list to string discontiguous memory blocks together to improve the utilization of pinned memory blocks.

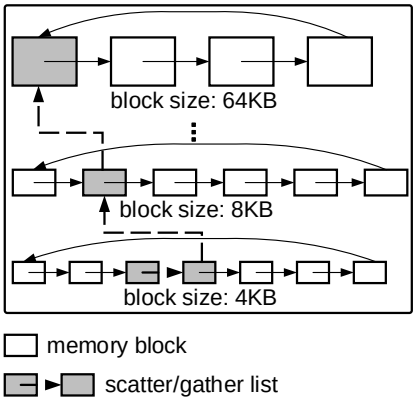

Figure 4. The pinned memory pool design.

4.3.1. Pinned Memory Pool Allocation

We dynamically allocate and pin memory blocks for I/O requests, but we do not release blocks of memory that have been pinned immediately after the DMA operation. Instead, we recycle the used and pinned memory blocks into the pinned memory pool for management. The pseudocode of the algorithm is shown in Algorithm 1. The input information is the size of the pinned memory area the application needs to allocate. The output information is the first address information of the memory area found in the pinned memory pool using the algorithm designed in this section. Algorithm 1 needs to deal with three possible situations. The first is when the pinned memory pool does not exist or is empty. The second is when the pinned memory pool is not empty, but the available pinned memory does not meet the demand. The third is when the pinned memory pool is not empty, and the available pinned memory can meet the demand.

For the first situation, we create a pinned memory pool when the application initializes the NVMe SSD's usage environment, shown in lines 1–6 of Algorithm 1. The pinned memory pool is empty at this stage, causing the application to initiate the first NVMe SSD's access request. When the applicant applies for memory allocation for the first time, we allocate and pin a memory block according to the memory block size configured by the user. After that, we put the memory block into the index for the next memory I/O request (see line 3). We mark the memory area as used, update the memory block's meta information, and return the first address of the memory area to the application. When this pinned memory is used up, we change its state to unused and manage it using a linked list rather than unpinning and releasing it directly.

For the second and third situations, we do not immediately apply for and pin new memory blocks because there may be blocks of memory that meet the requirements in the pool. Alternatively, we first find the pinned memory pool and get the information through the index of the linked list, shown in lines 8–17 of Algorithm 1. The core idea is to find out whether there is a pinned memory block that meets the memory pool requirements. Moreover, we implement a new pinned memory block management method based on scatter/gather lists in find_memory_region. As shown in Figure 4, benefiting from the scatter/gather lists, we can connect discrete pinned memory blocks to construct a large block. If enough blocks can meet the demand, we directly fetch the corresponding memory from the memory pool instead of allocating and pinning a new memory block. At this time, the time cost of pinning memory is saved. However, if the difference cannot find pinned memory that meets the requirements, we allocate and pin a new memory block, shown in lines 18–22 of Algorithm 1.

Algorithm 1 Pinned memory pool allocation

```
Input: memory_size
Output: memory_address
 1: if memory_pool is null then
 2:    memory_chunk ← alloc_and_pin(pin_size)
 3:    add_memory_pool(memory_pool, memory_chunk)
 4:    memory_address ← find_memory_region(memory_chunk)
 5:    update(memory_chunk)
 6:    return memory_address
 7: else
 8:    for memory_chunk in memory_pool do
 9:       if mem_free_size(memory_chunk) > memory_size then
10:          memory_address ← find_memory_region(memory_chunk)
11:          update(memory_chunk)
12:          return memory_address
13:       end if
14:    end for
15:    if total_memory_size + pin_size > total_pin_size then
16:       return null
17:    end if
18:    memory_chunk ← alloc_and_pin(pin_size)
19:    memory_address ← find_memory_region(memory_chunk)
20:    add_memory_pool(memory_pool, memory_chunk)
21:    update(memory_chunk)
22:    return memory_address
23: end if
```

The memory blocks used by traditional DMA must be contiguous, so if the free memory blocks in the pinned memory pool are large enough but not contiguous, we cannot use the free memory. Instead, we must allocate and pin new blocks of memory. This reduces the utilization of memory blocks and increases the I/O latency by allocating and pinning new memory blocks. We implemented a pinned memory management method based on the scatter/gather lists to solve this problem. The scatter/gather lists concatenated the discontiguous memory blocks in the pinned memory pool. Thus, we could assign them to serve user-level DMA transfers.

4.3.2. Pinned Memory Pool Release

We designed a pinned memory block release algorithm, and the pseudocode is shown in Algorithm 2. The input is the first address of the pinned memory area to be released by the application. The output is the memory block's status, representing the status of the execution result of the release process so that the upper application can judge the execution result. The upper-layer application may mistakenly pass in a memory address that does not exist in the pinned memory pool, shown in lines 2–4 of Algorithm 2.

We traverse the linked list of pinned memory blocks, finding out if there are contiguous, mergeable memory blocks, shown in lines 9–20 of Algorithm 2. The accessible memory areas before and after merging prevent fragmented memory space when releasing the memory area. If we find blocks that can be merged, we first merge them to get a larger contiguous pinned memory space and then mark it as free. If such memory blocks are not found, we directly mark them as free. After completing the marking of the memory blocks that can be released, as shown in lines 21–24 of Algorithm 2, we detect whether the new time of these memory blocks exceeds the threshold set by the user. Only the memory blocks that exceed the new time set by the user are released.

Algorithm 2 Pinned memory pool release

Input: memory_address
Output: status

```
memory_chunk ← find_memory_chunk(memory_pool, memory_address)
if memory_chunk is null then
    return status::error
end if
pin_memory_free(memory_chunk, memory_address)
update(memory_chunk)
mem_pre_region ← chunk_pre_region(memory_chunk, memory_address)
mem_cur_region ← memory_address
while memory_pre_region is free do
    merge_free(memory_chunk, mem_pre_region, mem_cur_region)
    mem_cur_region ← mem_pre_region
    mem_pre_region ← chunk_pre_region(memory_chunk, mem_cur_region)
end while
mem_next_region ← chunk_pre_region(memory_chunk, memory_address)
mem_cur_region ← memory_address
while memory_next_region is free do
    merge_free(memory_chunk, mem_next_region, mem_cur_region)
    mem_cur_region ← mem_next_region
    mem_next_region ← chunk_next_region(memory_chunk, mem_cur_region)
end while
if memory_chunk is free and memory_chunk_time_interval > time then
    delete_memory_chunk(memory_pool, memory_chunk)
    free_and_unpin(memory_chunk)
end if
return status::ok
```

4.4. Parameters Setting

In uDMA, there are some parameters to set. They are: (1) m, that is, the size of the pinned memory block for each application, (2) M, the total size of the memory cache area, and (3) t, the time threshold when the memory block is not accessed.

m can be set according to the application load characteristics or empirical values. The default value of m is 4 MB. We can adjust the pinned memory block size according to the memory block usage of the application during a period. After the application is executed for some time, we judge the memory usage of the current application and then adjust the memory block size appropriately to improve the performance.

There is no specific policy for setting M, because the more significant the value is set at, the better the concurrency of I/O request processing. However, it needs to be set reasonably according to different situations. For example, there is no other application to run on a server that only does disk I/O. Users can set a considerable value to improve the processing efficiency of I/O requests. The setting principle is to set a significant threshold without affecting the operation of other applications.

The primary consideration of the parameter of t is to release some completely free memory blocks and reduce the occupation of memory space when I/O processing is less in a period. The value of t is also a practical value, which is closely related to the application environment. If the value of t is too large, some of the memory may be pinned for a long time and not in use, resulting in a waste of memory resources. If the value of t is too small, it may result in frequent allocating and pinning memory.

5. Experiments and Results

In this section, we evaluate the efficiency of the proposed uDMA. First, we introduce the experimental setup including the experimental environment and benchmarks. Then,

we conduct several evaluation schemes to test the I/O performance. Finally, we analyze the performance of uDMA.

5.1. Experimental Setup

Table 1 shows the hardware and software environment of the experiment. We conducted a comparative experiment on a physical server with an Intel Xeon (R) Gold 5115 CPU. The CentOS 7.4 operating system and 20.01 SPDK were installed on the server. All performance evaluations were conducted on a commercial Intel Optane 900P Series NVMe SSD with a 280 GB capacity.

Table 1. The configuration of the experiment.

Environment	Configuration
CPU	Intel(R) Xeon(R) Gold 5115 CPU @ 2.40 GHz
Memory	128 G
OS	CentOS Linux release 7.4.1708 (Core)
SPDK	20.01
GCC	4.8.5
NVMe Storage	Intel Optane 900P Series 280 GB NVMe SSD

We compared the performance of uDMA with the latest user-level DMA methods implemented in SPDK [13]. We tested the effectiveness of uDMA under two benchmarks: a microbenchmark, perf [27,28], and a real-word benchmark, RocksDB [29]. perf is a benchmarking tool that can be used for performance tests. It has been widely used to test I/O performance [13,14]. RocksDB is a key–value and high-performance embedded database. It implements an LSM-tree storage engine to provide a key–value store and reading/writing functions. The configuration of perf and RocksDB were consistent with the official test of SPDK.

5.2. Micro Benchmark Test

We used perf to generate several I/O requests for different data sizes: (1) 50,000 random I/O requests for data less than or equal to 4 KB, (2) 5000 random I/O requests for medium data with sizes of 4 KB to 128 KB, and (3) 5000 random I/O requests for big data sizes larger than 128 KB. The memory block size was also set to three cases. The blocks used for small IO requests were set to 32 MB, the pinned memory block size was 4 MB, and the memory block size used for data larger than 128 KB was set to 32 MB.

Figure 5 shows the average response time of I/O requests for different data sizes. The x-axis and the y-axis represent the size of I/O requests and the average response time in seconds. It can be seen from Figure 5 that the performance of the NVMe SSD improved under all memory sizes tested, and the performance improvement was more evident under different data sizes. The main reason was that the algorithm's key idea was to allocate the pinned memory overhead to more requests and reduce the overhead of memory-related operations. It also optimized small data requests and used memory copy to effectively reduce the pinned memory overhead. The pinned memory pool was constructed in memory blocks for medium data to allocate the pinned memory overhead to multiple I/O requests. It could also be seen that under the same data size, with the increase of the program running time, the longer the memory usage time, the more pronounced the improvement effect of the algorithm. This was because with the increase of memory usage time, each memory block was used by more requests, and the pinned memory overhead of each memory block was allocated to more requests, which reduced the proportion of the overall memory operation-related overhead.

Figure 6 shows the performance improvement effect of the algorithm under various test data sizes. It can be seen from the figure that under most data sizes, with the increase of the program running time, the improvement effect of the algorithm became more prominent.

The main reason was that more requests used the same memory block, and its pinned memory overhead was allocated to more requests accordingly.

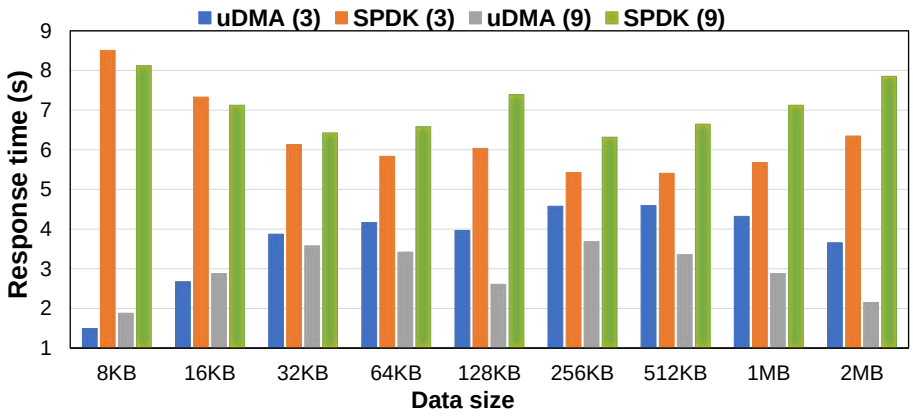

Figure 5. I/O response time under different data sizes.

The second experiment verified the algorithm's efficiency under different program running times. The maximum sizes of selected memory test data were 64 KB and 128 KB. The test results are shown in Figure 7. It can be seen from the figure that with the increase in the running time of the simulation program, the effect of the algorithm was more prominent. The main reason was that with the increase in the program running time, the memory library in SPDK applied for and pinned memory every time and released memory after use, which introduced a significant overhead. The algorithm allocated the pinned memory overhead to more I/O requests, reducing the total memory operation overhead. The results showed the effectiveness of the algorithm. Allocating the allocated memory overhead to more requests could effectively improve performance.

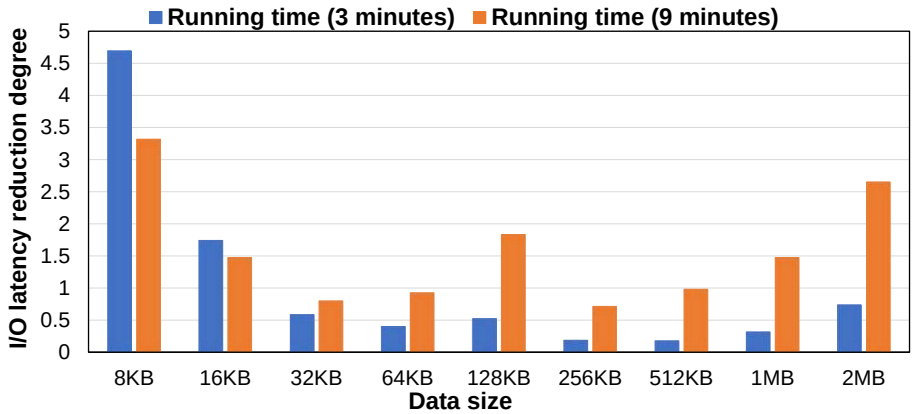

Figure 6. Performance improvement under different program running time.

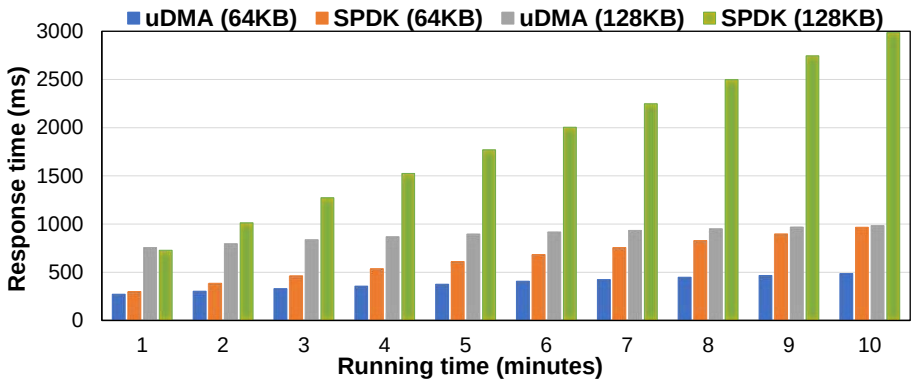

Figure 7. Influence of running time on Algorithm 1.

5.3. Real-World Benchmark Test

We ran the db_bench implemented in RocksDB to test the effectiveness of uDMA. To get closer to the natural environment, we used the ReadRandomWriteRandom mode to test the ops and micros of RocksDB under uDMA and SPDK. We ran RocksDB for 5 min per test, and the block size gradually increased from 4 KB to 128 KB.

Figure 8 shows the average I/O latency of RocksDB under uDMA and SPDK. Compared with SPDK, uDMA reduced the I/O latency by an average of 11.13%. According to the results shown in the figure, we can see that when the data size was 4 KB, the performance of uDMA was similar to that of SPDK. However, with the gradual increase of the data size, the reduced I/O latency of uDMA increased gradually, with a maximum of 11.39%. The main reason was that uDMA used fixed memory pools to reduce user-level DMA requests and fixed memory operations. By contrast, SPDK required frequent allocations and pinned memory blocks.

The IOPS results under SPDK and uDMA are shown in Figure 9. The IOPS of both SPDK and uDMA decreased as the data size increased. In the initial phase, when the data size was 4 KB, the IOPS of SPDK and uDMA were similar. However, with the gradual increase in data size, the decrease of the IOPS of SPDK was significantly higher than that of uDMA. By contrast, uDMA's IOPS was up about 9.5%. Because SPDK needed to allocate and pin memory blocks frequently for I/O requests, its IOPS degradation was relatively large.

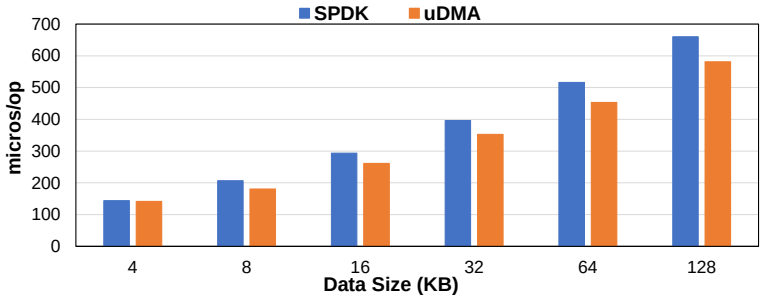

Figure 8. I/O latency under different data sizes.

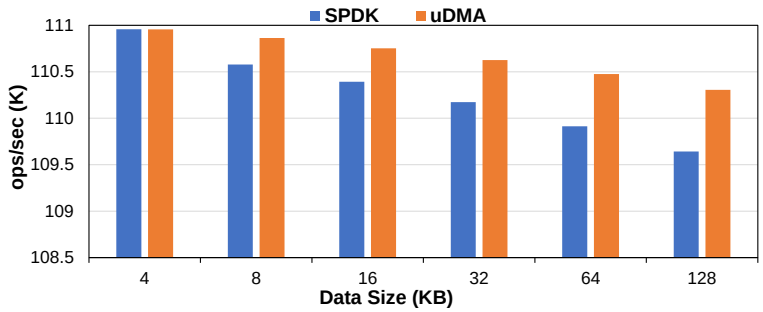

Figure 9. IOPS under different data sizes.

6. Conclusions

The user-mode I/O frameworks introduce additional overhead, such as pinning memory, which is a very time-consuming operation for I/O-intensive applications. This paper proposed an alternative and efficient memory management strategy named uDMA. uDMA can amortize per-request latency by dynamically selecting the appropriate pinned memory mode for different sizes of I/O requests. Moreover, uDMA uses the scatter/gather lists to improve the utilization of pinned memory blocks. The experimental results verified the effectiveness of the proposed uDMA. Our future research will further focus on improving user-mode I/O frameworks' performance. Combining the user-mode I/O scheme with some new characteristics of NVMe SSDs (e.g., open-channel SSDs) is a promising research direction.

Author Contributions: Conceptualization, L.W. and L.X.; Methodology, J.Z., L.W. and L.X.; Resources, L.X.; Writing—original draft, J.Z.; Supervision, G.Q. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by by the National Natural Science Foundation of China under grant no. 62104014, the National Natural Science Foundation of China under grant no. 62272026, the National Laboratory of Software Development Environment under grant no. SKLSDE-2022ZX-07.

Data Availability Statement: Data will be made available on request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kc, K.; Hsu, C.J.; Freeh, V.W. Evaluation of MapReduce in a Large Cluster. In Proceedings of the 8th International Conference on Cloud Computing, New York, NY, USA, 27 June–2 July 2015.
2. Dong, S.Y.; Kryczka, A.; Jin, Y.Q.; Stumm, M. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *Acm Trans. Storage* **2021**, *17*, 1–32. [CrossRef]
3. Chen, C.L.P.; Zhang, C.Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Inf. Sci.* **2014**, *275*, 314–347. [CrossRef]
4. NVM Express Workgroup. NVM Express Base Specification, Revision 2.0. Available online: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-5.pdf[2021] (accessed on 1 September 2022).
5. Li, F.F. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proc. Vldb Endow.* **2019**, *12*, 2263–2272. [CrossRef]
6. Xu, Q.; Siyamwala, H.; Ghosh, M.; Suri, T.; Awasthi, M.; Guz, Z.; Shayesteh, A.; Balakrishnan, V. Performance Analysis of NVMe SSDs and their Implication on Real World Databases. In Proceedings of the 8th International Systems and Storage Conference (SYSTOR), Haifa, Israel, 26–28 May 2015; pp. 1–11.
7. Intel. Intel Solid State Drive 750 Series. Proceedings of the VLDB Endowment. Available online: https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf (accessed on 1 September 2022).
8. Huo, Z.S.; Xiao, L.M.; Guo, M.Y.; Rong, X. Incremental Throughput Allocation of Heterogeneous Storage with No Disruptions in Dynamic Setting. *IEEE Trans. Comput.* **2020**, *69*, 679–698. [CrossRef]
9. Huo, Z.; Guo, M.; Xiao, L.; He, Z.; Rong, X.; Wei, B. TACD: A throughput allocation method based on variant of Cobb–Douglas for hybrid storage system. *J. Parallel Distrib. Comput.* **2019**, *128*, 43–56. [CrossRef]
10. Wang, Y.; Huang, J.F.; Chen, J.; Mao, R. PVSensing: A Process-Variation-Aware Space Allocation Strategy for 3D NAND Flash Memory. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2021**, *41*, 1302–1315. [CrossRef]
11. Alibaba Group. In-Storage Computing SSD Specifications and Applications. Available online: snia.org/sites/default/files/computational/20190808_COMP302A-1_Qui.pdff[2019] (accessed on 1 September 2022).
12. Amazon Web Services. Maximizing Microsoft SQL Server Performance Using Amazon EC2 NVMe Instance Store. Available online: https://d1.awsstatic.com/whitepapers/maximizing-microsoft-sql-using-ec2-nvme-instance-store.pdf[2020] (accessed on 1 September 2022).
13. Yang, Z.; Harris, J.R.; Walker, B.; Verkamp, D.; Liu, C.; Chang, C.; Cao, G.; Stern, J.; Verma, V.; Paul, L.E. SPDK: A Development Kit to Build High Performance Storage Applications. In Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 11–14 December 2017.
14. Yang, Z.; Liu, C.; Zhou, Y.; Liu, X.; Cao, G. SPDK vhost-NVMe: Accelerating IOs in virtual machines on NVMe SSDs via user space vhost target. In Proceedings of the 8th International Symposium on Cloud and Service Computing (SC2), Paris, France, 18–21 November 2018.
15. Kim, H.J.; Kim, J.S. A user-space storage IO framework for NVMe SSDs in mobile smart devices. *IEEE Trans. Consum. Electron.* **2017**, *63*, 28–35. [CrossRef]
16. Zhu, J.; Xiao, L.; Wang, L.; Qin, G.; Zhang, R.; Liu, Y.; Liu, Z. UPM-DMA: An Efficient Userspace DMA-Pinned Memory Management Strategy for NVMe SSDs. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*; Springer: Cham, Switzerland, 2021; pp. 257–270.
17. Kim, H.J.; Yee, Y.S.; Kim, J.S. NVMeDirect: A User-space IO Framework for Application-specific Optimization on NVMe SSDs. In Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), Denver, CO, USA, 20–21 June 2016.
18. Yang, Z.Y.; Wan, Q.; Cao, G.; Latecki, K. uNVMe-TCP: A User Space Approach to Optimizing NVMe over Fabrics TCP Transport. In Proceedings of the Internet of Vehicles. Technologies and Services Toward Smart Cities, Kaohsiung, Taiwan, 18—21 November 2019.
19. Available online: https://spdk.io/doc/memory.html (accessed on 1 September 2022).
20. Yu, Y.J.; Shin, D.I.; Shin, W.; Song, N.Y.; Choi, J.W.; Kim, H.S.; Eom, H.; Yeom, H.Y. Optimizing the Block I/O Subsystem for Fast Storage Devices. *Acm Trans. Comput. Syst.* **2014**, *32*, 1–48. [CrossRef]

21. Song, N.Y.; Song, Y.S.; Han, H.; Yeom, H.Y. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Trans. Storage* **2016**, *12*, 1–27. [CrossRef]
22. Lee, J.; Kim, Y.; Shipman, G.M.; Oral, S.; Kim, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2013**, *32*, 1–14. [CrossRef]
23. Wang, M.Y.; Hu, Y.M. An I/O scheduler based on fine-grained access patterns to improve SSD performance and lifespan. In Proceedings of the 29th Annual ACM Symposium on Applied Computing, Gyeongju, Republic of Korea, 24–28 March 2014; pp. 1511–1516.
24. Kim, J.; Seo, S.; Jung, D.; Kim, J.S.; Huh, J. Parameter-Aware I/O Management for Solid State Disks (SSDs). *IEEE Trans. Comput.* **2011**, *61*, 1–15.
25. Guz, Z.; Li, H.; Shayesteh, A.; Balakrishnan, V. Performance Characterization of NVMe-over-Fabrics Storage Disaggregation. *Acm Trans. Storage* **2018**, *14*, 1–18. [CrossRef]
26. Tian, K.; Zhang, Y.; Kang, L.; Zhao, Y.; Dong, Y. coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct IO. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, 15–17 July 2020.
27. Available online: https://spdk.io/ (accessed on 1 September 2022).
28. System Performance Tools: Perf. Available online: http://daslab.seas.harvard.edu/classes/cs165/doc/sections/S8_perf.pdf (accessed on 1 September 2022).
29. Dong, S.Y.; Kryczka, A.; Jin, Y.Q. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In Proceedings of the 19th USENIX Conference on File and Storage Technologies, Virtual Event, 23–25 February 2021; pp. 33–49.