NUMA-Aware Thread Migration for High Performance NVMM File Systems

Ying Wang, Dejun Jiang and Jin Xiong
SKL Computer Architecture, ICT, CAS; University of Chinese Academy of Sciences
{wangying01, jiangdejun, xiongjin}@ict.ac.cn

***Abstract*—Emerging Non-Volatile Main Memories (NVMMs) provide persistent storage and can be directly attached to the memory bus, which allows building file systems on non-volatile main memory (NVMM file systems). Since file systems are built on memory, NUMA architecture has a large impact on their performance due to the presence of remote memory access and imbalanced resource usage. Existing works migrate thread and thread data on DRAM to solve these problems. Unlike DRAM, NVMM introduces extra latency and lifetime limitations. This results in expensive data migration for NVMM file systems on NUMA architecture. In this paper, we argue that NUMA-aware thread migration without migrating data is desirable for NVMM file systems. We propose NThread, a NUMA-aware thread migration module for NVMM file system. NThread applies what-if analysis to get the node that each thread performs local access and evaluate what resource contention will be if all threads access data locally. Then NThread adopts migration based on priority to reduce NVMM and CPU contention. In addition, NThread also considers CPU cache sharing between threads for NVMM file systems when migrating threads. We implement NThread in state-of-the-art NVMM file system and compare it against existing NUMA-unaware NVMM file system ext4-dax, PMFS and NOVA. NThread improves throughput by 166.5%, 872.0% and 78.2% on average respectively for filebench. For running RocksDB, NThread achieves performance improvement by 111.3%, 57.9%, 32.8% on average.**

***Index Terms*—non-volatile memory, NUMA architecture, file system, performance, thread migration**

I. Introduction

Non-Uniform Memory Access (NUMA) architecture is widely used in data center [40], [43], [51]. They provide DRAM on each NUMA node with multi-core CPU, which can enlarge DRAM capacity. Emerging byte-addressable Non-Volatile Main Memories (NVMMs), such as Phase Change Memory (PCM) [3], [37], ReRAM [5] and recent Optane DC persistent memory [20], can be directly attached to memory bus meanwhile supports data persistency. Thus, one can build file systems on NVMM (namely NVMM file systems). Similar to DRAM, NVMM can also be structured on NUMA architecture to exploit multiple CPUs and enlarge NVMM capacity.

However, NUMA architecture introduces remote memory access in which applications run on one NUMA node may access data placed on another node. In addition, imbalanced request scheduling and data requests on NUMA nodes lead to imbalanced resource usage and further result in resource contention [9], such as memory accessing contention. Existing NVMM file systems [7], [10], [11], [23], [25], [45], [47], [48] are unaware of NUMA architecture. They place file data without considering the NVMM usage on NUMA nodes. Besides, application threads accessing file system rely on the default operating system thread scheduler, which migrates thread only considering CPU utilization. These bring remote memory access and resource contentions to application threads when reading and writing files, and thus reduce the performance of NVMM file systems. We observe that when performing file reads/writes from 4 KB to 256 KB on a NVMM file system (NOVA [47] on NVMM), the average latency of accessing remote node increases by 65.5 % compared to accessing local node. The average bandwidth is reduced by 34.6%. Besides, the imbalanced NVMM accessing increases file read latency by 73.0%. Thus, NVMM file systems are required being NUMA-aware to achieve better performance.

A number of research efforts have been made to improve application performance on NUMA architecture, such as reducing remote memory access [6], [38], reducing DRAM accessing imbalance [19], [38], and increasing CPU cache sharing among threads [28], [44]. The key idea of these works is migrating threads as well as related data (such as stack data and heap data on memory). These techniques are efficient for DRAM-based NUMA architecture. DRAM has high bandwidth and low latency. The cost of migrating thread data on DRAM is low. However, applying these techniques to NVMM file systems on NUMA architecture is expensive. Firstly, NVMM has higher access latency and lower bandwidth than DRAM. It is expensive to migrate file data on NVMM. For example, as reported in [22], the write bandwidth of NVMM is almost 1/6 of DRAM. In our observation, migrating a 16 KB page from one NVMM NUMA node to another takes 2.8x longer than that on DRAM. Secondly, unlike thread runtime data, migrating file data requires modifying file metadata to record the new addresses of data blocks. Besides, since file metadata changes, one needs to pay for extra effort to guarantee crash consistency, such as recording journal for metadata changes. These further increase the overhead of migrating file data for NVMM file systems. Thirdly, NVMM has lower write endurance than DRAM, migrating file data on NVMM introduces additional write operations and reduces device life. Finally, the stack and heap data of a thread are usually excluded from other threads. The data of stack and heap can be migrated only considering the corresponding thread. However, file data can be shared among multiple threads. Migrating file data based on the current state of a single thread is inaccurate and may cause migration oscillation. Taking the

key-value database RocksDB as an example, RocksDB usually adopts multiple threads to accelerate performance. Two threads may query key-value items from the same file (in the form of SSTable file) simultaneously. It is difficult to decide file data migration in case of the two threads running on different NUMA nodes.

Thus, we argue that one should only carefully migrate threads without migrating file data on NVMM file systems to reduce remote memory access and meanwhile avoid imbalanced resource usage. Recently, a few works [39], [46] propose to only migrate threads to the NUMA node where the accessing file locates to reduce remote access. However, they do not handle imbalanced NVMM accessing and CPU utilization on NUMA architecture. Furthermore, these works require modifying the application code. For example, [46] modifies application to invoke two additional system calls to obtain the location of file data and migrate threads respectively. Since file systems contain file related information, such as file data location and file data sharing, we can directly let file system migrate thread without modifying application code.

In this paper, the basic principle of thread migration is to migrate threads to the NUMA node where the accessing file locates. In such doing, remote memory access can be reduced. However, the basic principle faces three challenges. Firstly, a thread may access multiple files that are placed on different NUMA nodes. For example, a RocksDB thread may search multiple files (SSTable) to find a key value pair. In this case, carefully thread migration is required to avoid migration oscillation. Secondly, the number of threads on a NUMA node may increase after thread migration, which may cause imbalanced CPU usage and further result in CPU contention. Besides, NVMM file systems may place file data unevenly, which may cause imbalanced NVMM accessing and further result in NVMM contention (in this paper, all contention is caused by imbalanced use of resource). Migrating threads to reduce remote access may further exacerbate NVMM contention. To reduce resource contention, one needs to conduct thread migration by balancing reducing remote access and reducing resource contention. Finally, in case of multiple threads accessing the same file data, running these threads on different NUMA nodes cannot benefit from sharing the last level cache (LLC). One still needs to balance the benefit of reducing resource contention and increasing LLC sharing.

We propose NUMA-aware thread migration (NThread) for NVMM file systems. NThread applies what-if analysis to evaluate what resource contention will be if all threads access data locally. NThread obtains the node that performs local data access for each thread according to the read amount on each NUMA node. NThread also takes NVMM contention and CPU contention into account. By analyzing the impacts of different contention, NThread adopts priority based migration policy. The policy by default lets all threads access data locally in case of no resource contention. It takes NVMM contention as the first priority and migrates threads with high write ratio to remote nodes to reduce NVMM contention. Then the policy considers CPU contention and migrates threads to other nodes to avoid CPU contention. In case of multiple threads accessing the same file data, NThread keeps all threads locally to increase CPU cache sharing among threads. We implement NThread on the existing NVMM file system NOVA [47] under Linux kernel 4.18.8 and compare it with state-of-the-art NVMM based file systems. The evaluation shows that compared to NOVA, NThread increases throughput by 78.2% and 32.8% on average for filebench and RocksDB respectively.

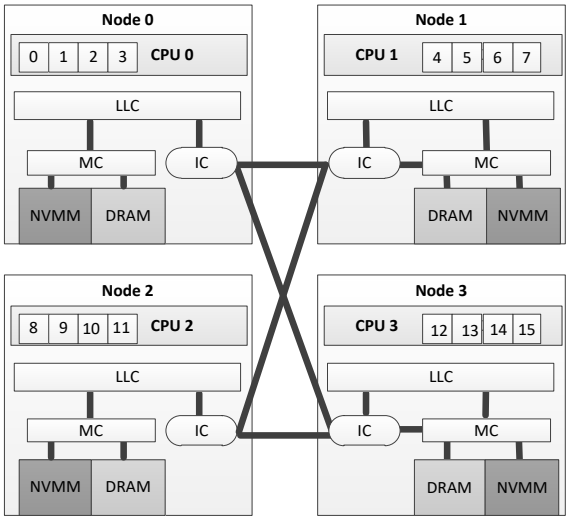


Fig. 1. The architecture of NUMA with 4 nodes. Each node has one CPU, and each CPU contains four cores.

II. Background and Motivation

A. NUMA architecture

NUMA architecture has multiple NUMA nodes and multiple CPU sockets. Each CPU socket contains multiple CPU cores (e.g. 4 cores per CPU socket in Figure 1). These cores share CPU last level cache (LLC). A CPU socket connects to one local NUMA node by Memory Controller and one or multiple remote NUMA nodes by Interconnect Network (IC in Figure 1). All NUMA nodes provide a single globally-addressable physical memory space with support for cache coherence [50]. In NUMA-based systems, accessing remote NUMA node suffers from higher latency than accessing local NUMA node. This also results in IC contention. Moreover, imbalanced data accessing may result in resource contention on a specific NUMA node. For example, intensive writes on one NUMA node brings NVMM contention to the node.

B. Non-volatile main memory

Emerging Non-Volatile Main Memories (NVMMs) provide persistency, byte-addressability and directly access features. We can build file system on it (NVMM file system). Since NVMM provides less write endurance (10^6-10^8) than DRAM (10^{16}) [11], [36], operating on NVMM file systems should avoid introducing extra write operations, such as repeated write operations by migrating file data. Recently, Intel provides a NVMM product – Intel Optane DC Persistent

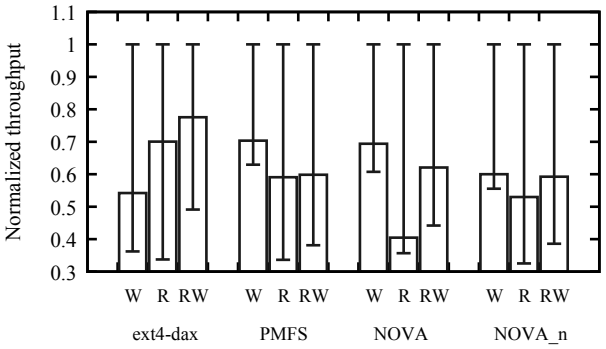


Fig. 2. The normalized throughput against maximum throughput. We run two RocksDB applications with db_bench. Each db_bench contains 10 threads. *W*, *R* and *RW* represent fillseq, readrandom and readwhilewriting respectively.

Memory Module (Optane DC PMM). The read latency of Optane DC PMM is about 3x slower than DRAM and the write latency is closed to DRAM [22]. For read and write bandwidth, Optane DC PMM is 1/3 and 1/6 of DRAM respectively. In addition, the read bandwidth of Optane DC PMM is 3x the write bandwidth. Similar to DRAM, NVMM is also affected by NUMA architecture, which includes remote access and imbalanced resource usage.

C. Motivation

Existing Linux operating systems use first-touch or interleave approaches to allocate memory space and randomly schedule threads on NUMA architecture. First-touch allocates space on the node where the thread runs, which allows threads to access data in local node, reducing remote access. However, this approach can cause memory accessing contention when a NUMA node holds a large volume of data. More importantly, operating systems may schedule threads to different nodes. Thus, first-touch may suffer from resource contention and remote access. Interleave approach allocates space by using round robin on all NUMA nodes. This helps to reduce memory accessing contention but increases remote access. These two approaches mainly optimize memory space allocate, such as memory for heap, stack and page cache. However, NVMM file systems usually bypass page cache [7], [10], [11], [23], [25], [41], [45], [47] and use its own space allocator. This requires the space allocator of NVMM file system to deal with the problems on NUMA architecture.

Existing NVMM file systems are not aware of NUMA architecture. They allocate space without considering hardware resources on NUMA nodes. Besides, they rely on kernel thread scheduler to select CPU node for running threads, in which only CPU utilization is considered but file data locations are ignored. This results in remote accessing when a thread and its file data are not on the same node. We run experiments to show the performance degradation of existing NVMM file systems with two-nodes NUMA architecture. The detailed experiment configurations are presented in Section V. Since existing NVMM file systems do not support multiple nodes and can only store files on a single NUMA node. We build one file system on each node and run a RocksDB instance with each file system separately. For comparison, we add multiple NUMA nodes support in NOVA to show the performance of existing NVMM file system on multiple NUMA nodes (NOVA_n). NOVA_n treats NVMM devices on multiple NUMA nodes as a single device. For example, *pmem0* and *pmem1* are the NVMM device of node 0 and node 1 respectively. NOVA only operates one NVMM device (such as *pmem0*) by mapping it into kernel space. Instead, NOVA_n maps the two NVMM devices into the kernel together as a continuous address space. NOVA_n randomly writes file data on NUMA nodes. We select three RocksDB workloads, including fileseq (W), readrandom (R) and readwhilewriting (RW) from db_bench to show the performance. The database size is 40 GB for each RocksDB run. We run the evaluation ten times.

Figure 2 shows the normalized throughputs of two applications. The max value of each error bar represents the achieved maximum throughput when all RocksDB threads are pinned to the node where the data file locates. This allows all data accessing to be performed locally. The min value of each error bar represents the achieved minimum throughput when all data accessing is performed remotely. In this case, all RocksDB threads run on a node from the one the data file locates. Existing NVMM file systems are NUMA-unware, and IO threads are placed by the kernel thread scheduler without considering NUMA architecture. The average throughputs are achieved under these cases. Compared to the maximum throughputs, the average throughputs of ext4-dax, PMFS, NOVA and NOVA_n reach 77.6%, 59.8%, 62.0% and 59.2% of the maximum ones respectively under readwhilewriting workload. Since threads are scheduled without considering data placement, existing NUMA-unaware NVMM file systems suffer from remote access and degraded performance. This motivates us to explore NUMA-aware approach for NVMM file systems.

III. Design

In this section, we present the design issues of NThread, a NUMA-aware thread migration module for NVMM file systems. The key idea of NThread is to migrate threads without migrating file data. This can reduce remote access and imbalanced resource usage as well as avoids expensive data migration on NVMM.

When NVMM file systems are mounted, NThread starts working. Note that, NThread does not decide the initial node where a thread runs, which is still decided by the operating system thread scheduler. Operating system thread scheduler determines that where a thread runs based on the CPU utilization on each node. It may result in remote access on NUMA architecture. Besides, it cannot solve NVMM contention. NThread periodically (such as 1s) runs what-if analysis to migrate threads to reduce remote memory access and resource contention.

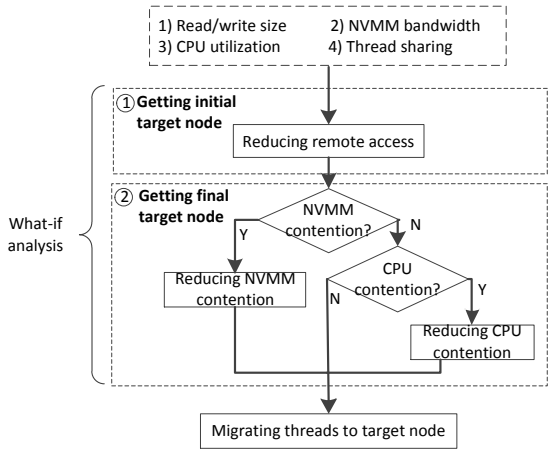


Fig. 3. The workflow of NThread each period.

Figure 3 shows the workflow of NThread each period. NThread firstly collects runtime information of each thread and node, including the data size of read/write, NVMM accessing bandwidth, CPU utilization and thread sharing information. Then NThread performs what-if analysis to get the target node where a thread should be migrated to. After what-if analysis, NThread migrates threads to the target node to improve performance. What-if analysis can directly decide the target node by reducing remote access. However, migrating all threads to perform local access cannot reduce NVMM contention and may result in CPU contention. For example, all threads read file data on node 0, and migrating all these threads to node 0 can cause CPU contention on node 0. Thus, what-if analysis decides the target node with two steps. Firstly, what-if analysis decides the initial target node to reduce remote access (① in Figure 3, Section III-A). Secondly, what-if analysis adopts priority based target node selection (② in Figure 3) to get the final target node to reduce NVMM and CPU contention (Section III-C). Note that, migrating threads to reduce CPU/NVMM contention may suffer from extra remote access again. However, we do not take this into account as resource contention has larger impact on performance degradation. In case of existing severe NVMM contention, increasing remote access turns out to improve overall performance [29]. Meanwhile, migrating threads to reduce resource contention may destroy CPU cache sharing, which can affect data accessing performance. Thus, what-if analysis also considers handling data sharing among threads (Section III-D).

A. Deciding initial target node

The basic principle of what-if analysis is to assume that all threads access data locally. For a given thread, we decide the initial target node to reduce remote access.

For file write, NThread always avoids remote write. NThread lets the thread directly write data on the NUMA node where threads run. As illustrated in Section III-C, a writing thread may be migrated to other nodes for reducing resource contention. Even under such a case, the writing thread still executes local writes.

For file read, a thread could access different file data and the file data can be distributed on different NUMA nodes. For example, one thread can read multiple SSTables to get target key-value pair, and these SSTables are stored on different NUMA nodes. File systems provide data read APIs with the data size. Reading a large amount of data usually indicates intensive read pressure. Thus, for one thread reading data of multiple files, NThread decides its initial target node according to its read amount on each NUMA node. To reduce thread migration oscillation, only when the read data size of a thread on one NUMA node is higher than all other nodes by a value per period (such as 200 MB, the bandwidth of one Optane DC PMM is approximately 6.8 GB/s), we think the node has a lot of data to read for the thread and regard the node as initial target node to reduce remote access. Otherwise, we think that the thread does not have an initial target node. Migrating threads to the initial target node can reduce remote memory access.

B. Resource contention analysis

When deciding the initial target node, NThread primarily considers reducing remote access. However, it cannot reduce NVMM contention and may cause CPU contention on NUMA architecture. Thus, NThread needs to take NVMM contention and CPU contention into account for deciding the final target node. Before presenting the policy for selecting the final target node, we analyze the different impacts of NVMM contention and CPU contention to decide which contention is more important.

On one hand, the contention comes from NVMM file system itself which places file data without considering the usage of hardware resources on NUMA architecture. If the access of NVMM is imbalanced among NUMA nodes, it will lead to amounts of hot data on a node, resulting in serious NVMM accessing contentions on that node. Our results show that NVMM contention can reduce performance by 73.0%. On the other hand, when multiple threads run on the same node, these threads compete for CPU resources. CPU contention can reduce performance by 39.3%.

In order to evaluate the impact of both NVMM and CPU contention. We conduct experiments to analyze them. All threads and file data are operated on node 0. To show the impact of CPU contention, we set some cores offline by keeping 4 cores online on node 0 and run more threads (8 and 16). To reflect the performance impact of reducing CPU contention, We migrate threads to node 1 and keep the data operation of the migrated threads on node 0. Figure 4 shows the normalized bandwidth against without handling CPU contention. The detailed experiment configurations are presented in Section V. For *1KB_8*, *1KB* represents read/write block size and *8* represents the number of running threads. *rw80* presents file read ratio is 80%. When performing 1 KB

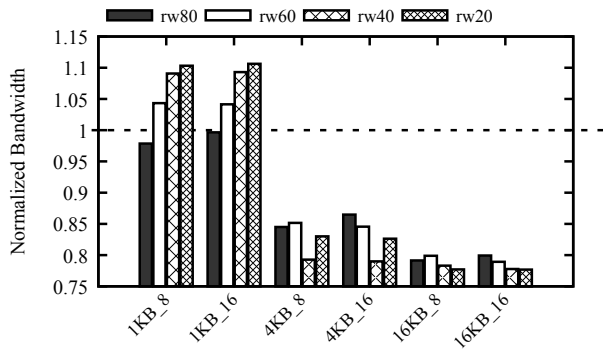


Fig. 4. The performance of reducing CPU contention.

TABLE I
NOTATIONS USED IN NODE SELECTION OF NTHREAD

BWr_N	The read bandwidth of Node N.
BWw_N	The write bandwidth of Node N.
BW_N	The total read and write bandwith of Node N.
$BW_{average}$	The average NVMM bandwidth of all nodes.
BWr_{weight}	The weight of read bandwidth: 1/3 in this paper.
D_write_i	The data size of file write for thread i within each time window.
D_read_i	The data size of file read for thread i within each time window.
W_ratio_i	The write ratio of thread i.
$D\_migrate_N$	The amount of data that Node N can be migrated.
CPU_i	The CPU utilization of thread i.
CPU_N	The CPU utilizaiton of Node N.
CPU_{limit}	The upper limit of CPU utilization: 90% in this paper.
$Time_i$	The time that thread i runs within each time window.
$Time_W$	The execution time window of NThread, such as 1s.

read/write operations, the evaluation does not cause NVMM contention (*1KB_8* and *1KB_16*) and migrating threads to other free nodes can reduce CPU contention. Figure 4 shows that the bandwidth can be improved by 13.2%. When increasing access size (4KB/16KB), theoretically reducing CPU contention can improve performance. However, NVMM has serious contention, and the performance decreased. The results show that the performance can be reduced by 20%. This is because NVMM contention is the main reason for performance. Therefore, when there exists both NVMM accessing contention and CPU contention, NThread only handles NVMM contention (see Figure 3).

C. Priority based target node selection

Only deciding target node by reducing remote access cannot solve resource contention problems. Therefore, what-if analysis also uses priority based migration policy to decide final target node. What-if analysis considers NVMM contention and CPU contention. When there exists both NVMM contention and CPU contention, NThread only handles the problem of NVMM contention (as shown Section III-B). The notations used in node selection are listed in Table I.

1) Avoiding NVMM accessing contention: We can migrate a part of hot data from high NVMM accessing contention node to low NVMM accessing contention node to solve this problem. However, NVMM provides higher access latency and lower bandwidth than DRAM, migrating file data between NVMM of different nodes is expensive and increases additional access pressure. Our results show that migrating 16 KB data on NVMM costs 35 us, which is 2.8x higher than DRAM. File systems contain a large number of files and a file can be accessed by multiple threads. It is difficult to maintain the accessing status of each file and migrate them. Therefore, NThread does not migrate file data. In addition, if NThread let a thread that runs on high NVMM accessing contention node write new data on low NVMM accessing contention node, it can reduce NVMM contention on high NVMM accessing contention node but introduce remote write operations. Remote writing has higher latency than local writing. Our results show that remote writing reduces performance by 65.5% compared to local writing. Besides, remoting write may result in remote read of these new writing data, increasing remote access. Therefore, NThread migrates threads and writes file data to the node where the thread runs to reduce NVMM accessing contention.

NThread obtains NVMM bandwidth for each node (we use ipmwatch [21] to get the bandwidth) to detect if imbalanced NVMM accessing occurs. NThread calculates the total NVMM bandwidth of each node by Equation 1. The BWr_N and BWw_N are the read and write bandwidth of Node N. BW_N is the bandwidth of Node N after our calculation. BWr_{weight} is the weight of read bandwidth. Since the read bandwidth of NVMM is higher than its write bandwidth, we reduce the weight of read bandwidth. In this paper, we choose the 1/3 as the BWr_{weight}. We get this value from the evaluation in Section V-G. NThread judges the imbalanced NVMM accessing by considering the theoretical maximum bandwidth of Optane DC PMM[1] and our test results. We can get the maximum bandwidth of a node by multiplying the theoretical maximum bandwidth of one Optane DC PMM by the number of Optane DC PMM. In this paper, when the NVMM bandwidth of a node exceeds 80% of the maximum bandwidth and the NVMM bandwidth of other nodes is lower than 1/2 of the node, we argue that the NVMM accessing is imbalanced and the node has high NVMM accessing contention. NThread should migrate threads to balance NVMM accessing.

$$BW_N = BWr_N * BWr\_{weight} + BWw_N \quad (1)$$

NThread performs NVMM balancing operations by migrating threads with high write ratio from the high NVMM accessing contention node to low NVMM accessing contention node. Since NThread sets file write position on the node where the application thread runs, moving away high write ratio

[1]the read and write bandwidth of single Optane DC PMM are approximately 6.8 GB/s and 1.85 GB/s respectively and the bandwidth of multiple devices is superimposed.

threads can reduce NVMM write operations on high NVMM accessing contention nodes. Besides, writing more new data on low NVMM accessing contention nodes can bring new file data to the node, introducing read operations. This approach avoids file data migration overhead of NVMM and solves the NVMM accessing contention problems. NThread calculates write ratio of each thread by Equation 2. *D_write*$_i$ and *D_read*$_i$ are the data size of file write and read for thread i within each time window (the running cycle of NThread, *Time*$_W$, such as 1s) respectively. Note that, NThread currently does not support *mmap* operations. The *D_write*$_i$ and *D_read*$_i$ are measured only for read/write system calls.

$$W\_ratio_i = D\_write_i/(D\_write_i + D\_read_i) \quad (2)$$

Since migrating threads may introduce new NVMM contention, such as migrating too many threads to low NVMM accessing nodes, NThread evaluates the amount of data that can be migrated for migrating threads. Equation 3 shows the calculation formula. BW$_N$ is the NVMM bandwidth of Node N and BW$_{average}$ is the average of NVMM bandwidth of all nodes. D_migrate$_N$ is the amount of data that Node N can be migrated in (D_migrate$_N$ is less than 0) or out (D_migrate$_N$ is greater than 0). We multiply the value of bandwidth by time window (Time$_W$, the running cycle of NThread, such as 1s) because we perform balancing operations per time window.

$$D\_migrate_N = (BW_N - BW_{average}) * Time_W \quad (3)$$

Migrating a thread to low NVMM accessing contention node should satisfy Equation 4. This limitation ensures that the data amount of NVMM accessing on Node N does not exceed BW$_{average}$ after migrating thread i to it, avoiding introducing new contention on Node N. Since migrating thread i to Node N only brings write operations to Node N, we use the amount of write data for thread i (D_write$_i$). Once Equation 4 is satisfied, node N is the final target node of thread i.

$$D\_migrate_N + D\_write_i <= 0 \quad (4)$$

2) Avoiding CPU contention: NThread migrates threads to reduce remote memory access and NVMM accessing imbalance. However, these operations can bring CPU contention if we do not consider CPU utilization. NThread calculates CPU utilization to avoid CPU contention caused by imbalanced CPU usage. Since NVMM contention has greater impact than CPU contention (Section III-B), NThread handles CPU contention only when the system does not have NVMM contention.

NThread calculates the CPU utilization for each thread and sum the CPU utilization of each NUMA node. When NThread finds that NUMA nodes have imbalanced CPU utilization that the utilization of one CPU is 2x of the other CPUs and the CPU utilization exceeds CPU$_{limit}$ (we set the CPU$_{limit}$ is 90% and get this value through experiments in Section V-G), NThread migrates threads to balance the usage of CPU. NThread migrates one thread to a node only if the CPU utilization of the node does not exceed a threshold after migrating. As shown in Equation 5, NThread migrates thread i to node N only if the sum of the CPU utilization of node N and thread i does not exceed CPU$_{limit}$. CPU$_N$ is the utilization of CPU on Node N, which is the sum of the CPU utilization of all threads running on Node N. CPU$_i$ is the CPU utilization of thread i. Once Equation 5 is satisfied, node N is the final target node of thread i. We calculate CPU utilization for each thread according to Equation 6. *Time*$_i$ is the time that thread i runs within each time window. *Cores* is the number of core in each physical CPU. *Time*$_W$ is the time window of NThread (such as 1s). This approach can avoid serious CPU contention within each CPU.

$$CPU_i + CPU_N <= CPU_{limit} \quad (5)$$

$$CPU_i = Time_i/(Time_W * cores) \quad (6)$$

D. Handling data sharing

All threads performing local access can let threads that access the same data run on the same node, increase CPU cache sharing among threads. However, reducing resource contention by migrating threads (Section III-C) may migrate threads that access the same data to different NUMA nodes, destroying CPU cache sharing between threads. CPU cache sharing can avoid reading data from NVMM and improve performance. Our results show that CPU cache sharing among threads can improve performance by 31.5% when reading file data. Therefore, NThread avoids migrating a thread to reduce resource contention if the thread accesses the same file with other threads simultaneously.

NThread obtains threads sharing information from virtual file system (VFS). VFS records the number of threads that each file is being accessed on (i_count in inode). When a thread read/write a file and the i_count of the file is greater than 1, the thread is sharing file data with other threads. NThread does not migrate the thread during the process of reducing resource contention.

IV. Implementation

NThread is a module for NVMM file systems which periodically (such as 1s) fetches information from NVMM file systems and performing thread migration. We skip some threads, such that read/write some fewer data, to optimize the process. NThread is transparent to applications. We implement NThread based on NOVA [47] on Linux kernel 4.18.8. NThread can be used for all NVMM file systems. We modify file data block index in NOVA to let NThread get the node where accessing file data is located for each thread. Besides, we modify file system space allocator to support allocating space on a specified node. Therefore, each thread can write data on the node where the thread is running on. In NThread, we migrate thread by using binding function in operating system (*sched_setaffinity*). In total, implementing NThread requires total 1,300 lines of codes.

TABLE II
THE CONFIGURATION OF NTHREAD. *Y/N* REPRESENT ENABLING/DISABLING THE FEATURE.

	Reducing remote access	Reducing CPU contention	Reducing NVMM contention	Increasing data sharing
NThread_rl	Y	N	N	N
NThread_cpu	Y	Y	N	N
NThread_nm	Y	Y	Y	N
NThread	Y	Y	Y	Y

V. EVALUATION

In this section, we evaluate NThread and show the performance improvement by reducing remote access, avoiding CPU, NVMM contention and increasing data sharing.

A. Experimental setup

We conduct all experiments on a server equipped with two NUMA nodes. Each node contains an Intel(R) Xeon(R) Gold 5215 CPU (2.50GH) processor, a 128 GB Optane DC PMM device and 64 GB DRAM. The operating system is CentOS 7.6.1810, Linux kernel version is 4.18.8. We configure Optane DC PMM with App direct mode [22]. All experimental results are the average of at least 3 runs.

B. Compared systems

We compare NThread with ext4-dax [8], PMFS [11] and NOVA [47]. Since Optane DC PMM cannot establish continuous regions across NUMA nodes, NVMM device (pmemX) is installed on each NUMA node. Ext4-dax, PMFS and NOVA only support building file system on one device, they cannot build a file system across multiple nodes. Therefore, we build a file system for each node. For single application test (fio, filebench and RocksDB), we just test one file system for them. For two applications, such as two RocksDBs and filebenches, we run one application on each file system. In this case, running multiple applications for existing NVMM file systems does not have NVMM contention problem. To show the performance of existing NVMM file system on multiple NUMA nodes, we modify NOVA to build a file system supporting multiple nodes (NOVA_n, see Section II-C).

Table II shows the four configurations used in the evaluation. NThread_rl shows the benefits of reducing remote access, which pins threads to the NUMA node where the reading file locates. Besides, NThread_rl lets a thread directly write data on the NUMA node where the thread runs. Therefore, NThread_rl reduces remote access. NThread_cpu not only pins threads as NThread_rl, but also reduces CPU contention as shown in Section III-C2. This can reduce CPU contention as well as reduce remote access. Based on NThread_cpu, NThread_nm further reduces NVMM contention (Section III-C1). Finally, NThread enables all these configured features.

C. Microbenchmark

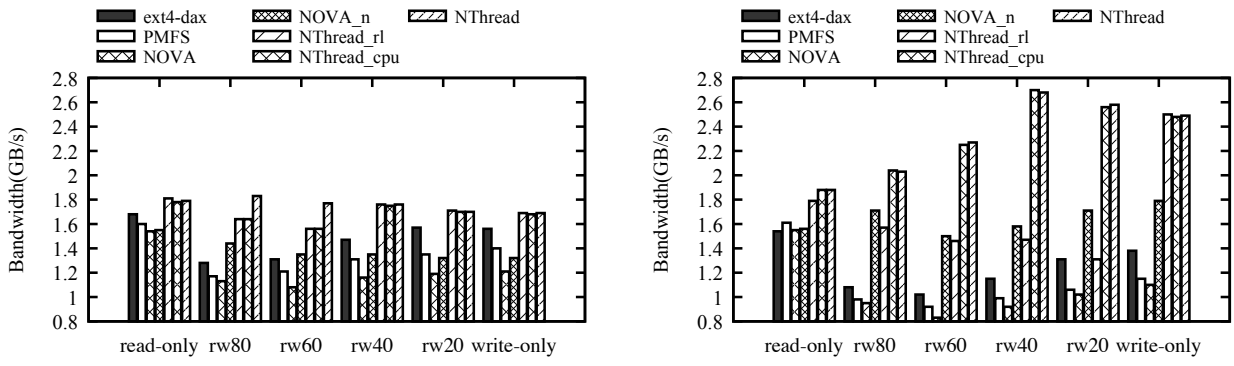
We use fio [2] to show the performance of NThread. Each thread accesses a 4 GB private file using 4 KB block size. We set file read ratio to 100% (read-only), 80% (rw80), 60% (rw60), 40% (rw40), 20% (rw20) and 0% (write-only). Since our server only contains one Optane DC PMM for each NUMA node, 4 running threads can run out of its bandwidth. We set each node with 4 active cores and set the rest offline. We run 4/8 threads to show the results with/without CPU contentions respectively. Figure 5 shows the bandwidth of each file system (we calculate the bandwidth by using Equation 1). Before the evaluation, we pre-allocate all file data on NUMA node 0 to show the performance of each file system.

When running 4 threads, as shown in Figure 5(a), NThread_rl improves bandwidth by 15.1%, 27.1%, 40.5% and 22.3% on average under all workloads against ext4-dax, PMFS, NOVA and NOVA_n respectively. This is because all file data is stored on node 0 and meanwhile NThread_rl binds all threads on node 0 to provide local file read without remote access. On the contrary, the 4 threads are interleaved on the two NUMA nodes by the kernel scheduler for existing NVMM file systems. This results in remote access and degrades performance. As each of the 4 threads runs on an individual core, NThread_cpu performs similar to NThread_rl due to few CPU contention.

When performing mixed read and write operations (rw80 and rw60), NThread improves performance by 12.6% and 12.7% on average against NThread_rl and NThread_cpu respectively. This is because NThread detects NVMM contention and migrates threads with high write ratios to other node. This in turn results in new data written to different nodes. Correspondingly, the reads to these data are distributed between two nodes, and thus reduces NVMM contention. As for rw40 and rw20, NThread does not detect NVMM contention and pins all threads on node 0, which is similar to NThread_rl. As for write-only workloads, there is no data reads. NThread_rl, NThread_cpu and NThread do not migrate threads to reduce remote access and resource contention. Thus, NThread performs similar to both NThread_rl and NThread_cpu. Currently, NThread does not address NVMM read contention. As a result, NThread performs similar to NThread_rl and NThread_cpu for read-only workload.

Figure 5(b) shows the results when running 8 threads, NThread increases bandwidth by 90.4%, 117.1%, 129,6%, 43.4% and 44.7% on average against ext4-dax, PMFS, NOVA, NOVA_n and NThread_rl respectively. Since the 8 threads compete for 4 cores, NThread addresses both CPU and NVMM contentions compared to NThread_rl. Moreover, unlike these existing file systems, NThread migrates threads to reduce remote access. This allows NThread to achieve bandwidth improvement. NThread_cpu performs similar with NThread. This is because initially the CPU and NVMM contentions occur on the same node 0, and NThread_cpu migrates 4 threads to node 1 to solve CPU contention. This in turn results in reduced reads/writes to node 0 as well as reduced NVMM contention on node 0.

As for read-only workloads, NThread_rl pins all threads on NUMA node 0, resulting in CPU and NVMM contention. Both NThread_cpu and NThread migrate threads to NUMA node 1



(a) Fio for running 4 threads

(b) Fio for running 8 threads

Fig. 5. The bandwidth of fio. *rw* presents read ratio.

to reduce CPU contention but still suffer from NVMM contention. Therefore, NThread_cpu and NThread only improve bandwidth by 5% compared to NThread_rl. For write-only workloads, NThread_rl, NThread_cpu and NThread directly write data on the node where the thread runs. Since there exist no remote accesses, the three approaches perform similarly. Therefore, they perform similar.

Note that, the threads of NOVA_n are evenly scheduled to run between two NUMA nodes. This can reduce both CPU and NVMM contentions, but still suffers from remote access under read-write mixed workloads. Thus, NOVA_n outperforms NThread_rl under read-write mixed workloads, but still performs worse than NThread_cpu and NThread.

NThread_rl shows lower bandwidth for running 8 threads (Figure 5(b)) than running 4 threads (Figure 5(a)) for read-only, rw80, rw60, rw40 and rw20 workloads. This is because NThread_rl binds all threads to NUMA node 0 to perform local data access. Since there exist only 4 active cores for each node, running 8 threads suffers from thread switching overheads. This reduces system performance. As for write-only workloads, write threads are evenly scheduled to each NUMA node for NThread_rl, which allows local writes. NThread_rl performs better for running 8 threads than running 4 threads.

D. Macrobenchmark

We use filebench [1], a file system benchmark that simulates a large variety of workloads, as macrobenchmark. Similar to previous works [34], [47], we use the default configuration to run filebench as show in Table III. We also let all CPU core online to show the performance. To show the performance of NThread under multiple applications running, we run filebench with one (Figure 6(a)) and two (Figure 6(b)) applications.

For running one application, NThread_rl outperforms ext4-dax, PMFS, NOVA and NOVA_n by 166.3%, 869.1%, 78.1% and 35.1% on average respectively. This is because NThread_rl avoids remote access. Besides, NThread_rl pins all threads to the NUMA node where the file data locates. It can increase data sharing between threads. NThread_nm and NThread_cpu

TABLE III
FILEBENCH WORKLOAD CHARACTERISTICS

Workload	Average file size	files	I/O size	threads	r:w ratio
Fileserver	128 KB	100 K	1MB	50	1:2
Webserver	16 KB	100 K	1MB	100	10:1
Webproxy	16 KB	100 K	16 KB	100	5:1
Varmail	16 KB	100 K	1 MB	16	1:1

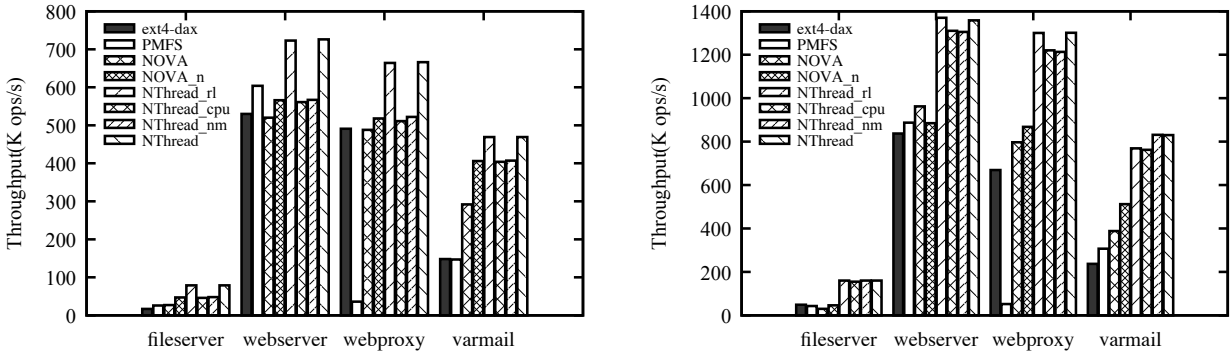
reduce resource contention but ignore data sharing between threads, they show poor performance than NThread_rl. NThread reads file counter to support file data sharing between threads and avoids migrating threads that accessing to the same data on different NUMA nodes. NThread performs better than NThread_nm and NThread_cpu.

For running two applications, existing NVMM file systems generate more remote access and resource contention. NThread_rl can make two applications run on different NUMA nodes, which does not produce remote access and resource contention. Therefore, NThread_rl improves performance by 157.5%, 729.7%, 160.8% and 103.0% on average for ext4-dax, PMFS, NOVA and NOVA_n. Because NThread runs an application on a NUMA node, the system does not have the problem of resource contention, NThread_cpu, NThread_nm and NThread show similar performance with NThread_rl.

E. Application

RocksDB [13] is a high-performance, persistent key-value store and is widely deployed for Internet services [18], [33], [42] and storage services [4], [14]–[16], [31]. RocksDB is implemented based on Log-Structure Merge Trees (LSM-Tree) [35] and relies on file system to support logs and data persistency. For example, RocksDB persists key-value operations in write-ahead log and flushes key-value pairs into Sorted String Table (SSTable) files on underlying file systems. Thus, RocksDB is widely used for evaluating file system performance [22], [27], [32], [46], [48], [49], [52]. Similarly, in this paper, we use RocksDB to evaluate the defectiveness of NThread.

We set RocksDB with 10 M keys and 4KB value size. The data size of each RocksDB application is 40 GB. To show the



(a) One filebench application

(b) Two filebench applications

Fig. 6. The throughput of filebench workloads

performance of NThread with multiple applications running, we run one (Figure 7(a)) and two (Figure 7(b)) RocksDB applications. Each RocksDB runs 10 threads, which is equal to the number of cores per CPU.

When running one RocksDB application, NThread improves throughput by 111.3%, 57.9%, 32.8% and 24.1% on average compared to ext4-dax, PMFS, NOVA and NOVA_n respectively. For fillseq (W) operations, NThread evenly writes all file data on the node where the thread runs, reducing remote write and NVMM contention. For readrandom (R) operations, NThread improves throughput by 68.0%, 50.1%, 50.4% and 12.8% for ext4-dax, PMFS, NOVA and NOVA_n respectively, as NThread reduces remote access. Since NOVA_n randomly writes data to each NUMA node, NVMM contention is avoided when performing read operations. Therefore, the performance of NOVA_n is better than ext4-dax, PMFS and NOVA. Readwhilewriting (RW) performs read and write operations, and NThread improves by 47.1%, 26.9%, 17.8% and 14.3% for ext4-dax, PMFS, NOVA and NOVA_n respectively.

For running two RocksDB applications, the remote access of existing NVMM file systems increases. NThread improves performance by 141.2%, 117.4%, 54.3% and 46.8% on average compared to ext4-dax, PMFS, NOVA and NOVA_n.

F. Multiple NUMA nodes

To show the performance of NThread under multiple memory nodes. We run NThread on a server with 4 NUMA nodes. Each node has an Intel (R) Xeon (R) Gold 5215 CPU processor and 16 GB DRAM. The operating system is CentOS 7.6.1810, Linux kernel version is 4.18.8. We use DRAM to emulate NVMM in this evaluation. Since DRAM space limits, we only run RocksDB with 1 M keys (the other configuration of RocksDB is the same as Section V-E). Figure 7(c) shows the results. We can see that NThread improves performance by 34.4%, 65.5%, 41.9% and 21.7% compared to ext4-dax, PMFS, NOVA and NOVA_n. Since we replace NVMM with DRAM in this evaluation and DRAM has high accessing bandwidth, the performance improvement of NThread is reduced.

G. Parameter Tuning

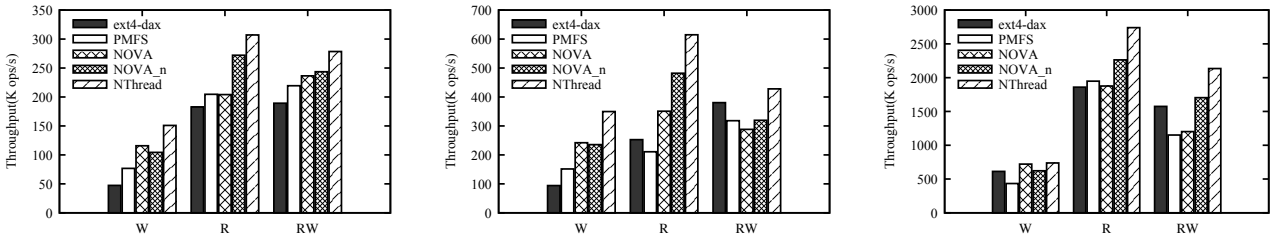
In this section, we show some parameters choices through evaluation. To show the impact of CPU contention, we set some cores offline by keeping 4 cores online on each node.

CPU utilization We choose CPU utilization setting CPU_{limit} in Equation 5 by migrating threads to node N and varying the value of CPU_{limit}. We set the value of CPU_{limit} to 80%, 90%, 100%, 110%, 130% and no limits respectively. We run the evaluation by using 16 threads. We set the operating size as 1 KB to avoid introducing NVMM contention. Figure 8 shows the normalized bandwidth against CPU_{limit} at 100%. When the CPU_{limit} is greater than 100%, CPU contention is increased and the performance is reduced. To reduce CPU contention and avoid remote access, NThread sets the CPU_{limit} as 90% in this paper.

NVMM contention Since the read bandwidth of Optane DC PMM is much higher than the write bandwidth, we cannot directly add read and write bandwidth to judge NVMM contention. We run evaluation to obtain the calculation approach of Optane DC PMM bandwidth. We mix read and write operations with read ratio from 100% to 0% (x-axis in Figure 9) and show the bandwidth (y-axis) under different calculation methods. When we calculate bandwidth by using 1/3 of read bandwidth, the bandwidth of Optane DC PMM has a stable value. Therefore, we set the weight of read bandwidth as 1/3 ($\text{BWr}_{\_weight}$ in Section III-C1). We calculate Optane DC PMM bandwidth by adding write bandwidth and 1/3 of read bandwidth. When the used bandwidth of one NUMA node exceeds 80% of the maximum bandwidth (Section III-C), we think that NVMM has severe access contention.

VI. Related work

Reducing remote access [12] builds a locality-aware page table, which provides accurate information to guide locality-aware thread and data mapping policies. However, it identifies the accessing thread within the page-table upon TLB miss, which requires hardware level facilities that are not available in all hardware. [26] builds temporal flows of interactions between threads and objects, which helps programmers understand why and which memory objects are accessed remotely.



(a) One RocksDB application (b) Tow RocksDB applications (c) Four RocksDB applications on four nodes

Fig. 7. The throughput of RocksDB. *W*, *R* and *RW* represents fillseq, readrandom and readwhilewriting respectively in db_bench.

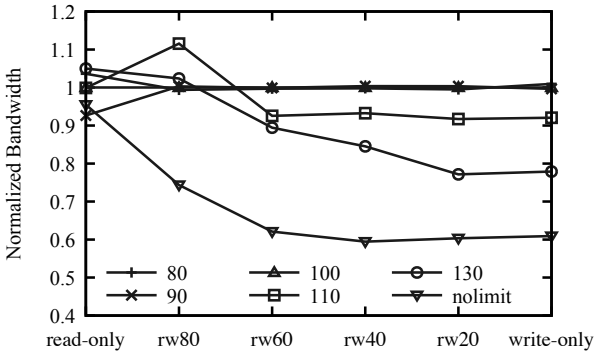


Fig. 8. CPU utilization

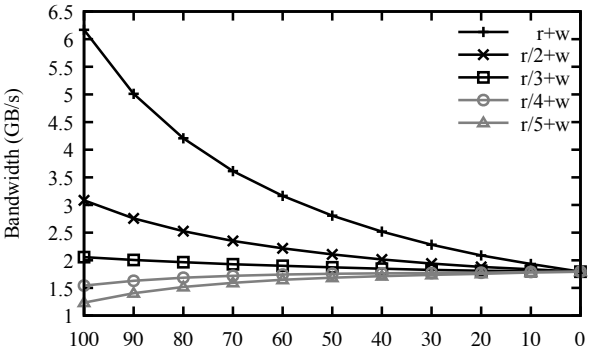


Fig. 9. NVMM bandwidth

NThread can directly discovery remote access by reading file system file block index. Operating systems allocate thread memory (such as stack and heap) on the node where the thread locates to avoid remote access. HydraFS [39] and [46] suggest binding threads to the NUMA node where the accessing file locates to reduce remote access. These works only reduce remote memory accessing on NUMA architecture. NThread also migrates thread based on the amount of data that the thread currently read to avoid remote access.

Resource contention There are many works solving resource contention. [6] uses LLC miss rate to predict contention between threads, and evenly allocates threads to each node by using LLC miss rate to avoid contention. However, LLC miss rate cannot reflect the CPU utilization and is likely to cause CPU contention. [44] predicts both memory bandwidth utilization and optimal core allocations with high accuracy and low overhead for memory-intensive multi-threaded applications on large-scale NUMA machines. [17] accounts for contention at multiple resources such as processor functional units and memory channels. And then it predicts the best thread allocation and resources needed for a given workload to meet a specified performance target. NThread predicts contention based on current resource usages, such as CPU utilization and NVMM bandwidth, which can better reflect the resource usage and contention. Then NThread dynamically migrates threads to improve performance.

[30] argues that maximizing data locality does not always minimize execution time, it may be more advantageous to allocate data on a remote processor to reduce memory contention. [9] solves memory contention on memory traffic by using page-replication, interleaving and co-location to allocate memory. [19], [28] consider the asymmetric interconnect architecture to maximize bandwidth and optimal page placement. All these works reduce resource contention by placing threads first and then migrate memory. However, NVMM has lower access latency and less bandwidth than DRAM. Migrating data on NVMM takes up limited bandwidth and reduces performance. NThread is designed for NVMM file systems, and it can reduce NVMM contention by specifying the write location of new file data. Therefore, NThread only migrates threads. [50] designs a bandwidth-aware memory placement policy to avoid memory contention. It places data amount according to the DRAM-to-NVMM bandwidth ratios. Since the bandwidth of DRAM is higher than NVMM, [50] places more data in DRAM than NVMM. However, the amount of data stored is not related with the access frequency of data. Placing less data on NVMM than DRAM does not mean NVMM has low access pressure.

Resource sharing [38] detects sharing patterns online with low overhead by using performance monitoring unit(PMU). [28] and [24] sample hardware counters to detect communicating threads and place them onto a well-connected nodes. NThread determines data sharing between threads by reading the status of file. If a file is accessed by multiple threads, the file is shared among these threads. And then NThread migrates these threads to the NUMA node where the file locates, supporting data sharing. NThread avoids migrating threads accessing the same data on different nodes.

Thread and memory migrating [28] migrates thread memory by using full memory migration and dynamic memory migration. [6] argues that migrating a large amount of thread memory with threads resulting in good performance. These works are orthogonal to NThread. NThread only migrates threads without memory migration in this paper.

VII. Conclusion

Since NVMM file systems directly access file data on the memory, NUMA architecture has a large impact on their performance due to the presence of remote memory access and resource contention. However, existing NVMM file systems, such as ext4-dax, PMFS and NOVA, are unware of NUMA architecture and thus suffer from degraded performance. In this paper, we avoid the expensive data migration on NVMM, and instead present NThread. NThread is a NUMA-aware thread migration approach for high performance NVMM file systems. NThread performs what-if analysis to find target node to reduce remote access and resource contention. Besides, NThread supports file data sharing between threads. Compared to existing NVMM file system NOVA, NThread achieves up to 205.6% and 44.6% throughput improvements for filebench and RocksDB applications.

Acknowledges

We thank the anonymous reviewers and our shepherd Andre Brinkmann for their insights and valuable comments. We also thank Alvaro Frank, Frederic Schimmelpfennig, Wanling Gao, Shukai Han and Wenqing Jia for their suggestions. This work is supported by National Key Research and Development Program of China under grant No.2016YFB1000302, Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDB44030200, Beijing Natural Science Foundation under grant No. L192038, and Youth Innovation Promotion Association CAS.

References

[1] Filebench 1.4.9.1. https://github.com/filebench/filebench/wiki.

[2] Fio-2.14. https://github.com/axboe/fio.

[3] A. M. Caulfield A. Akel, R. K. Gupta T. I. Mollov, and S. Swanson. Onyx: A protoype phase change memory storage array. *In Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'11, pages 2–2*, 2011.

[4] M. Annamalai. Zippydb: a modern, distributed keyvalue data store. https://www.youtube.com/watch?v=DfiN7pG0D0k, 2015.

[5] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 587–590, 2004.

[6] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1. USENIX Association, 2011.

[7] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Coetzee Derrick. Bpfs:better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146. ACM, 2009.

[8] Jonathan Corbet. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, September 2014.

[9] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394. ACM, 2013.

[10] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493. ACM, 2019.

[11] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15. ACM, 2014.

[12] Matthias Diener Eduardo H. M. Cruz, Laércio L. Pilla Marco A. Z. Alves, and Philippe O. A. Navaux. Optimizing memory locality using a locality-aware page table. *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, 2014.

[13] Facebook. Rocksdb. http://rocksdb.org/, 2013.

[14] Facebook. Cassandra on rocksdb at instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram., 2018.

[15] Facebook. Myrocks. http://myrocks.io/, 2019.

[16] S. Iyer G. J. Chen, J. L. Wiener, R. Lei A. Jaiswal, W. Wang N. Simha, T. Williamson K. Wilfong, and S. Yilmaz. Realtime data processing at facebook. *In Proceedings of the International Conference on Management of Data, pages 1087–1098. ACM*, 2016.

[17] Daniel Goodman, Georgios Varisteas, and Tim Harris. Pandia: Comprehensive contention-sensitive thread placement. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 254–269. ACM, 2017.

[18] A. Gupta. Followfeed: Linkedin's feed made faster and smarter. https://engineering.linkedin.com/blog/2016/03/followfeed–linkedin-s-feedmade-faster-and-smarter, 2016.

[19] David Gureya. Asymmetry-aware page placement for contemporary numa architectures. 2018.

[20] Intel. *Intel and Micron produce breakthrough memory technology*, https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/ edition.

[21] Intel. ipmwatch. https://github.com/opcm/pcm.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508. ACM, 2019.

[24] Ali Kamali. Sharing aware scheduling on multicore systems. *In MSc Thesis, Simon Fraser Univ.,*, 2010.

[25] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477. ACM, 2017.

[26] Renaud Lachaize, Baptiste Lepers, and Vivien Quema. Memprof: A memory profiler for NUMA multicore systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 53–64. USENIX, 2012.

[27] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.

[28] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289. USENIX Association, 2015.

[29] Zoltan Majo and Thomas R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10. ACM, 2011.

[30] Zoltan Majo and Thomas R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[31] Y. Matsunobu. Innodb to myrocks migration in main mysql database at facebook. USENIX Association, May 2017.

[32] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85. USENIX Association, 2016.

[33] S. Nanniyur. Sherpa scales new heights. https://yahooeng.tumblr.com/post/120730204806/ sherpa-scales-new-heights, 2015.

[34] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16. ACM, 2016.

[35] D. Gawlick P. O'Neil, E. Cheng and E. O'Neil. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.

[36] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[37] M. Breitwisch S. Raoux, G. Burr, R. Shelby C. Rettner, Y. Chen, D. Krebs M. Salinga, H. L. Lung S.-H. Chen, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development, 52(4.5):465–479*, 2008.

[38] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58. ACM, 2007.

[39] Kai Liu Ting Wu, Xianzhang Chen, Zhixiang Liu Chunhua Xiao, and Edwin H.-M. Sha Qingfeng Zhuge. Hydrafs: an efficient numa-aware in-memory file system. *Cluster*, 2019.

[40] Lingjia Tang ; Jason Mars ; Xiao Zhang ; Robert Hagmann ; Robert Hundt ; Eric Tune. Optimizing google's warehouse scale computers: The numa experience. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[41] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14. ACM, 2014.

[42] J. Wang. Myrocks: best practice at alibaba. https:/www.percona.com/live/17/sessions/myrocksbest-practice-alibaba, 2017.

[43] Junsheng Tan ; Fuzong Wang. Optimizing virtual machines scheduling on high performance network numa systems. *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, 2017.

[44] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *HPCA*, pages 419–431. IEEE Computer Society, 2016.

[45] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11. ACM, 2011.

[46] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439. ACM, 2019.

[47] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 323–338. USENIX Association, 2016.

[48] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496. ACM, 2017.

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[50] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 18:1–18:10. ACM, 2017.

[51] Yuxia Cheng ; Wenzhi Chen ; Zonghui Wang ; Xinjie Yu. Performance-monitoring-based traffic-aware virtual machine deployment on numa systems. *IEEE Systems Journal (Volume: 11 , Issue: 2 , June 2017)*, 2015.

[52] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, Boston, MA, 2019. USENIX Association.