2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C) | 978-1-6654-4393-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ACSOS-C52956.2021.00024

Is Data Migration Evil in the NVM File System?

Jungwook Han, Hongsu Byun, Hyungjoon Kwon, Sungyong Park and Youngjae Kim
Department of Computer Science and Engineering, Sogang University
Seoul, South Korea
{immerhjw, byhs, hishine6, parksy, youkim}@sogang.ac.kr

***Abstract*—The NVM file system often exhibits unstable I/O performance in a NUMA server environment due to frequent remote memory accesses when threads and data are exclusively placed on different NUMA nodes. Further, multiple threads may use all of the available bandwidth of the Integrated Memory Controller (iMC), causing an iMC bottleneck. `NThread` partly addresses the problems above by maximizing local memory accesses via migrating threads to data resident CPU node. However, `NThread` cannot benefit in cases when iMC is overloaded. Therefore, we propose `Dragonfly`, an approach that migrates data to the memory module of the CPU node where the thread is located when iMC is overloaded. The proposed approach inherently balances the load among iMCs, thus offering a fair load-balancing among iMCs. Specifically, `Dragonfly` implements a Migration Trigger Policy (MTP) to migrate data between CPU nodes on an opportunistic basis, minimizing the performance overhead caused by unnecessary data migration. We implement and evaluate `NThread` and `Dragonfly` in the NOVA file system deployed on an Intel Optane DC PM server for different application scenarios via Filebench workloads. The evaluation confirms that `Dragonfly` outperforms on an average 3.26× higher throughput than `NThread`.**

I. Introduction

The Intel Optane DC Persistent Memory (DCPM) server is a Non-Uniform Memory Access (NUMA) system with two CPU sockets and multiple non-volatile Optane DC Memory modules [1, 2]. Figure 1 illustrates the architecture of a NUMA-based Intel Optane DC PM server. The Optane DC Memory module is connected directly to the memory bus line and shares an integrated Memory Controller (iMC) with the DRAM. In the NUMA system, memory access can be classified into local memory access and remote memory access depending on the location of the threads and data [3, 4, 5]. According to the performance measurement of the DCPM server, remote access latency is about twice as large as local access latency due to the QPI link latency between CPU nodes [1, 6, 7, 8].

Several Non-Volatile Memory (NVM) file systems such as NOVA [9], Aerie [10], and SplitFS [11], Assise [12], and Strata [13] have been studied in the past few years. NOVA [9] is a state-of-the-art scalable log-structured NVM file system for manycore servers that ensures the consistency of data and metadata in the event of failures. NOVA implements per-inode logging for metadata and uses per-core data structures such as inode table, journaling space, and memory allocator and improves performance by supporting concurrent I/Os. NOVA achieves scalability for single file multiple write I/O patterns by adopting range lock instead of mutex lock on inode [14]. Similarly, in NUMA environments, NOVA increased scalability by adopting a NUMA-aware per-core data structure. However, NOVA suffers from unstable I/O response times due to local and remote memory accesses in the NUMA system [15].

Y. Kim is the corresponding author.

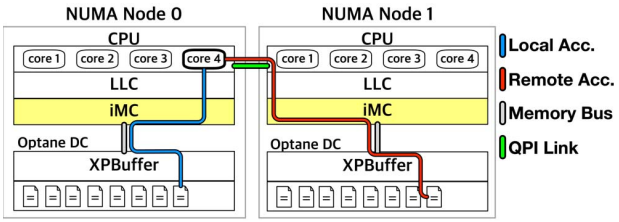

Fig. 1. An overview of NUMA-based Intel Optane DC PM server [1].

To address the aforementioned problems of NVM file system such as NOVA in the NUMA system, `NThread` [16] proposed to migrate I/O threads to the CPU nodes where the data is located. `NThread` uses an algorithm based on the iMC's load of the target node to which threads are migrated and determines whether or not to migrate them. If the target node's iMC is overloaded, `NThread` stops migrating threads. `NThread` uses the memory bandwidth history of the target node and predicts its load on the iMC. Based on this prediction, `NThread` precisely migrates the running thread to the target node only as long as its iMC is not overloaded. However, under such a target node's iMC overload situation, even though the running threads are not migrated to the target node, they still need to access data remotely through the overloaded iMC. `NThread` fails to balance the load on iMCs, resulting in overloaded iMC of the target node.

In such scenarios, `NThread` has the following problems:

- First, `NThread` must continue remote access because if the target node's iMC is overloaded, it will not migrate the thread location to where the data resides. This causes an increase in access latency as much as QPI link latency every time a thread accesses data [16].
- Second, the overloaded iMC is shared by threads running on the local CPU and the remote CPU. Therefore, there is performance interference between them, increasing their access latency.

To solve the aforementioned limitations of `NThread`, we propose `Dragonfly`, which opportunistically migrates data when the iMC of the target node to which a thread is migrated

978-1-6654-4393-7/21/$31.00 ©2021 IEEE
DOI 10.1109/ACSOS-C52956.2021.00024

is overloaded. Also, to minimize performance degradation and excessive write amplification problems due to indiscriminate data migration, `Dragonfly` employs a model-based Migration Trigger Policy (MTP). For evaluations, we implemented `NThread` and `Dragonfly` with the NOVA file system and evaluated them on an DCPM server running Linux v5.1.0 using Filebench workloads for various application scenarios. In our evaluations, `NThread` showed a 1.5×, 1.2×, and 7.1× increase in I/O throughput on Webserver, Webproxy, and Videoserver workloads compared to `NThread` respectively.

II. Background and Motivation

This section describes the Intel Optane DC PM Server architecture and presents our research motivation.

A. Intel Optane DC PM Server

The Optane DC Memory provides persistence, low access latency and high bandwidth comparable to DRAM. Figure 1 shows the architecture of the NUMA-based Intel Optane DC PM server. The server is composed of two CPU nodes, connected to each other via QPI links. Optane DC Memory is connected to the memory bus and exchanges data through iMC, like DRAM. Optane DC Memory has an XP-Controller in which there is a 16KB XP-buffer to improve read and write performance.

In the NUMA system, the remote access latency is greater than the local access latency because the QPI bandwidth between NUMA Nodes 0 and 1 is lower than the memory bus bandwidth. Also, iMC and XP-buffer can be overloaded when many threads access data from different NUMA nodes, which dramatically reduces the memory access speed of each thread. Figure 1 describes the remote access path and local access path. Remote access path shows that the thread running on Node0 accesses the file located in Node1 through QPI and the iMC and XPBuffer in Node1.

B. Motivation

As we discussed in Section I, there are two limitations to `NThread` in the NUMA system when the iMC is overhead. To illustrate these problems, we conducted the following experiments. We ran two Webserver (Light) applications on the NUMA server described in Table I in Section IV and measured the I/O throughput of each application. Before the experiments, files used by the application are fixed in the Optane DC Memory of Node0. We ran one application on Node0 and another one on Node1 Each application in this experiment is called Webserver (Node0) and Webserver (Node1). A detailed description of the Webserver (Light) application is presented in Table II in Section IV.

Since the files used by the application are fixed in the Optane DC Memory of Node0, Webserver (Node0) reads files locally, while Webserver (Node1) reads them remotely. We measured the throughput of Webserver (Node0) and Webserver (Node1) in situations where the iMC of Optane DC Memory in Node0 is not overloaded (Normal) and overloaded (iMC congested). To simulate a situation where the iMC is overloaded,

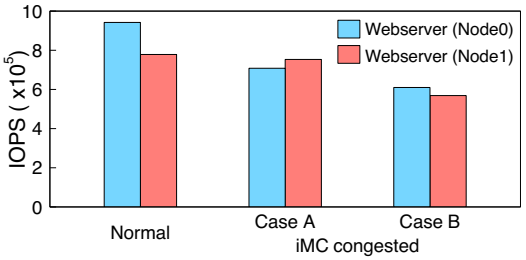

Fig. 2. Application throughput running on each node with and without overloaded iMC.

we ran three more dummy Webserver (Light) applications and fixed their files in the Optane DC Memory of Node0. In addition, to consider what is causing the overload of the iMC, we considered the case where three web server applications are run on Node0 (Case A) and the case where they are run on Node1 (Case B).

Figure 2 shows the throughput of each application – Webserver (Node0) and Webserver (Node1) for normal and iMC congested cases. In the case of Normal, the throughput of Webserver (Node0) and Webserver (Node1) are 972K IOPS and 778K IOPS respectively. On the other hand, where the iMC is overloaded, in case A, the throughput of Webserver (Node0) and Webserver (Node1) decreased by 24.8% and 3.3%, respectively, compared to Normal. In Case B, the throughput of Webserver (Node0) and Webserver (Node1) decreased by 35.3% and 27%, respectively, compared to Normal. Overall, no matter what overloaded the iMC, both Webserver (Node0) and Webserver (Node1) suffered significant performance degradation. When the overload of iMC occurs in Node0, Webserver (Node0) has severe performance degradation compared to Webserver (Node1).

In summary, the performance degradation of each application caused by iMC overload cannot be avoided by `NThread` if the data location is fixed. That is, `NThread` cannot solve the iMC overloading problem because it does not change the data location. In this paper, we propose a data migration technique, that distributes the overloaded iMC's load to another node's iMC and maximizes the local access of the application by reducing remote access.

III. Design And Implementation

This section describes the overview of `Dragonfly`, the implementation of `Dragonfly` in the NOVA file system, and the model-based migration policy.

A. Overview of `Dragonfly`

`Dragonfly` is a software module that performs data migration between NUMA nodes in NOVA. `Dragonfly` works in concert with `NThread`. If the iMC of the target to which the thread is to be migrated is overloaded, `NThread` does not attempt to migrate the thread. However, `Dragonfly` executes data migration, thus converting remote access to local access. Figure 3 shows the difference between `NThread` and `Dragonfly`. Figure 3(a) is the operation flow of `NThread`

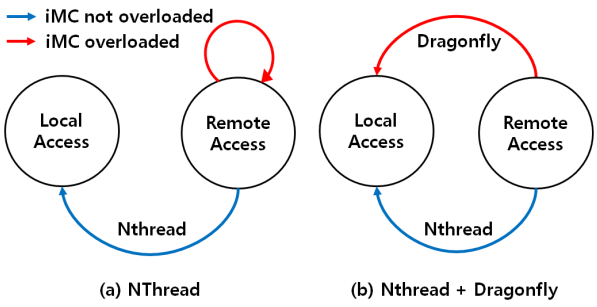

Fig. 3. State transition diagram of operation flow for `NThread` and `Dragonfly`.

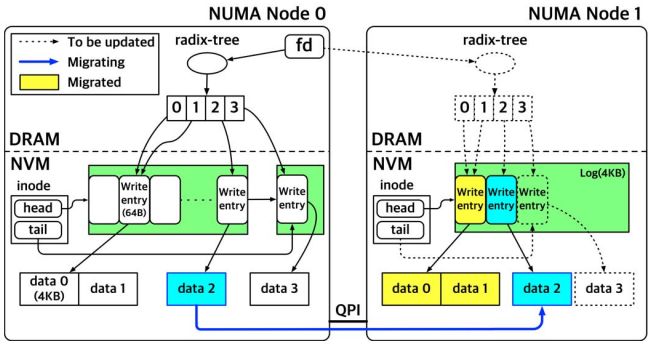

Fig. 4. Overview of NOVA file system and file migration from Node 0 to 1.

and Figure 3(b) depicts that when the target iMC is not overloaded during remote access, the operation flow is the same as that of `NThread`, but otherwise follows `Dragonfly`.

`Dragonfly` internally maintains a request queue that manages information about the files requested for data migration. `Dragonfly` inserts the file information of a thread that has failed to migrate due to the overload of the target node's iMC into the request queue. This request queue is periodically emptied. Data migration also runs in the background to minimize performance interference with foreground I/O. When executing each data migration, `Dragonfly` determines whether to migrate by calculating the migration opportunity cost according to the MTP. In addition, parameter settings such as T_{window} and T_m used in the MTP algorithm are described in detail in Section III-C. Ultimately, `Dragonfly` lowers the load on the overloaded iMC by migrating data instead of thread migration. Also, after data migration, threads access the data locally.

B. NOVA File System with `Dragonfly`

NOVA is a log-structure-based file system for NVM. NOVA has its own log for each file (inode) and log pages in 4KB units are connected and managed in a linked list form. In addition, NOVA stores meta data (64B) in the log and the actual data is stored in a data page (4KB). Data pages are linked to the log entry. In order to quickly access the data page of a file through a log entry, a radix tree for the file is maintained in DRAM.

Figure 4 depicts the NOVA file system architecture and how `Dragonfly` migrates data from Node0 to Node1 in the NOVA. Algorithm 1 describes the data migration process in the NOVA in detail. When migration starts, the migration state of the current file is allocated through an atomic operation (2, 18), and a new Inode is created in the dst Node through the super block (3). Copy the actual data page to dst Node through the $whileloop$ and create $write\_log\_entry$ for the data page and map $write\_log\_entry$ to the corresponding data pages(4 12). The write log entries and radix tree of the migrated file are reconstructed based on the migrated data pages (13, 14). In the process of mapping $filp$ (file pointer) to the newly created Inode (16), a lock is acquired with a spinlock to prevent any thread from accessing the file (15, 17). By resetting the valid bit of the old file in the src Node, the pages are returned and can be reused when a new file is allocated later (19). After changing $filp$, the application will be possible to access the new file created in the dst Node locally.

Moreover, in order to enable file access even during data migration, a shared lock (rw_semaphore) is used for the file structure. Files are migrated through the MTP, a policy that determines how many files and what files to migrate. A detailed description of the MTP is given in Section III-C.

C. Migration Trigger Policy

Performance gain can be expected by changing remote access to local access through data migration. However, if migration is performed without an appropriate policy, performance degradation and write amplification problems may occur [7, 8]. MTP was designed to maximize the performance gains from migration by solving the aforementioned problems. MTP checks the request queue at each time window and decides to migrate each file based on the model. `Dragonfly` can be triggered by system call with MTP at the user level, or can be performed by itself with MTP at the kernel level.

Algorithm 1 Migration Algorithm

```
1:  procedure MIGRATE(filp, src, dst)                    ▷ migrate file
2:      atomic_set(Inode_state, MIG)
3:      build new Inode in dst Node
4:      while numblock > 0 do                            ▷ migrate data pages
5:          read write_entry_log in src Node
6:          read according data_pages in src Node
7:          allocate new free_list in dst Node
8:          copied = memcpy_mcsafe(dst, src, pagesize)
9:          initialize write_log_entry for new data_pages
10:         append write_log_entry
11:         numblock -= copied
12:     end while
13:     update Inode_tail
14:     rebuild radix_tree                    ▷ for new write_log_entry
15:     spin_lock(f_lock)                              ▷ critical section
16:     Set filp to new Inode
17:     spin_unlock(f_lock)
18:     atomic_set(Inode_state, DONE)
19:     Invalidate old Inode
20:     return filesize
21: end procedure
```

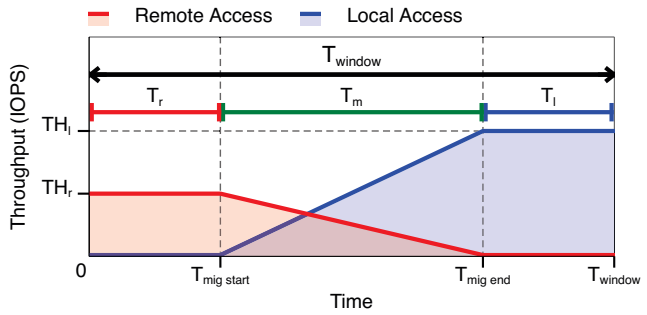

Fig. 5. Throughput by interval before, during, and after migration of data over time.

Currently, `Dragonfly` is implemented only in the former, and the latter will be implemented in future work.

Problem Formulation: MTP is a mathematical model-based data migration decision maker. Figure 5 describes the I/O throughput for each time section before, during, and after migrating data over time in one time window T_{window}.

To describe the mathematical model, the following notations are defined. Let T_r be the time section where only remote access occurs in time window T_{window}, T_m be the section where local access gradually increases during migration, and T_l be the section where local access occurs for all files after migration ends. That is, T_{window} is the sum of T_r, T_m, and T_l as follows:

$$T_{window} = T_r + T_m + T_l \tag{1}$$

Local access throughput and remote access throughput are denoted as TH_l and TH_r, respectively, and the migration overhead value is denoted as O_m. Here, O_m is the total amount of I/O data loss due to migration overhead during T_{window}, and the unit is byte. In Figure 5, TH_l and TH_r denote throughputs for each time section respectively.

In the case of migration, the throughput for a specific time t ($0 < t < T_m$) during T_m is equal to Equation (2).

$$\frac{1}{T_m}\{(T_m - t) \times TH_r + t \times TH_l\} \tag{2}$$

According to Equation (2), as time t passes, the remote access ratio decreases and the local access ratio increases. So total throughput at a specific time t increases.

From Equation (2), the total amount of data served during T_m is Equation (3) and the total amount of data served during T_{window} is Equation (4).

$$\int_0^{T_m} \{TH_r + \frac{TH_l - TH_r}{T_m} \times t\}dt - O_m \tag{3}$$

$$TH_r \times T_r + \int_0^{T_m} \{TH_r + \frac{TH_l - TH_r}{T_m} \times t\}dt - O_m + TH_l \times T_l \tag{4}$$

On the other hand, if data migration is not applied, all files in the request queue are remotely accessed, and the total amount of work for T_{window} is equal to Equation (5).

$$TH_r \times T_{window} \tag{5}$$

MTP compares Equation (4) and Equation (5) in each T_{window}. As seen in Equation (6), migration is triggered only when Equation (4) is greater than Equation (5). From Equation (6), the following Equation (7) is derived.

$$\int_0^{T_m} \{TH_r + \frac{TH_l - TH_r}{T_m} \times t\}dt - O_m + TH_l \times T_l > TH_r \times (T_m + T_l) \tag{6}$$

$$T_l > K - \frac{T_m}{2}, \quad (K = \frac{O_m}{TH_l - TH_r}) \tag{7}$$

In addition, T_r in Equation (7) does not act as a variable. Also, the remote access time T_r is almost zero because there is no significant overhead in the process of calculating the actual policy.

$$T_{window} \approx T_m + T_l, \ (\because (1), \ T_r \approx 0) \tag{8}$$

As a result, the following Equation (9) is derived.

$$T_m < 2 \times (T_{window} - K) \tag{9}$$

From Equation (9), we have the following analysis:

- First, when the application is running, K is set for the system dependent constants, TH_l, TH_r, and O_m. From Equation (9), T_{window} should be larger than K.
- Second, according to Equation (9) T_m must be smaller than $2 \times (T_{window} - K)$ to get throughput gain by data migration. In other words, if T_m is smaller than the threshold in T_{window}, it means that T_l is relatively greater by the expression (8). Increasing T_l leads to throughput gain by local access.

IV. Evaluation

This section describes the experimental setup and analyzes the evaluation results of `Dragonfly` for various realistic workloads.

A. Experimental Setup

Testbed Setup: We implemented `Dragonfly` in NOVA and evaluated on a real Intel Optane DC PM server running Linux kernel v5.1.0. The Intel Optane DC PM server has two 28 core NUMA Nodes, and each NUMA Node has 6 Optane DC modules. The sever specifications are shown in Table I.

Workloads and Application Scenarios: We performed the experiments using four workload applications (Webserver, Webproxy, Videoserver, and Fileserver) generated via Filebench [17]. Each application workload details are listed in Table II. Further, to clearly demonstrate the iMC overload

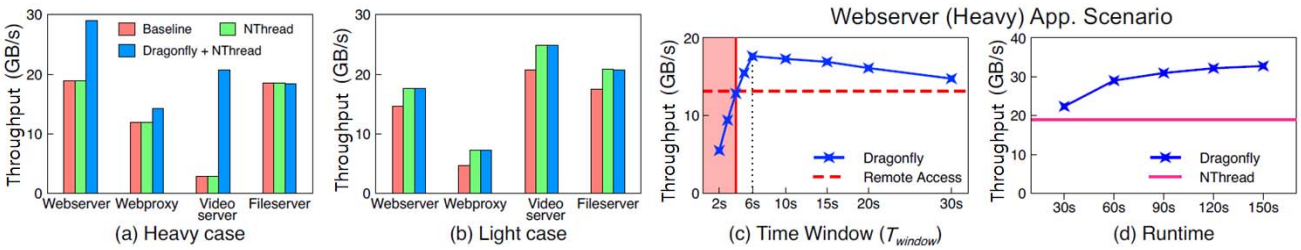

Fig. 6. Performance analysis of `Dragonfly`. Runtime of (a)-(c) is 60 seconds.

TABLE I
SPECIFICATIONS OF INTEL OPTANE DC PM SERVER.

CPU	Intel(R) Xeon(R) Platinum 8280M v2 2.70GHz CPU Nodes (#): 2, Cores per Node (#): 28
Memory	DRAMs per Node (#): 6, DDR4, 64 GB * 12 (=768GB)
PM	Intel Optane DC Persistent Memory PMs per Node (#): 6, 128 GB * 12 (=1.5TB)
OS	Linux kernel 5.1.0

TABLE II
FILEBENCH WORKLOAD CHARACTERISTICS. RO DENOTES READ ONLY.

Application	Heavy Case			Light Case			Read: Write Ratio
	Size (KB)	File (#)	Thr (#)	Size (KB)	File (#)	Thr (#)	
Webserver	160	10K	14	16	10K	7	10:1
Webproxy	160	100K	14	64	10K	7	5:1
Videoserver	2GB	50	28	1GB	50	14	RO
Fileserver	128	10k	14	128	1K	7	1:2

cases, we present two cases for each application workloads, i.e., Heavy and light.

Also, in `NThread`, both iMC load and CPU utilization of the target node were considered during thread migration, but in current scope of work, only iMC overload scenario was considered. Because, when I/O size is greater than 4KB, iMC overload has a higher impact on system throughput than CPU utilization. In our experiment, 16KB was set as the minimum I/O size. At 16KB, iMC overload was dominant rather than CPU utilization.

To simulate the situation of multiple applications running on a single server, we executed all four applications on the server for each application scenario of Webserver, Webproxy, and Fileserver workloads. For the Videoserver application scenario, only two applications run on the server. In this experiment, Note that, for this experiment, the location of the file significantly impacts the application performance.

Initial Data Placement: Files were initially fixed in the Optane DC Memory of Node0 before application execution. The reason for pre-allocating the initial data to one node is to simulate the effect of data migration. When multiple applications are executed, they are executed in either Node0 or Node1 to perform both local or remote memory access.

We compare the `Dragonfly` with the following variants:

- **Baseline:** Vanilla NOVA
- **`NThread`:** NOVA performing thread migration
- **`NThread` + `Dragonfly`:** NOVA performing data migration in addition to thread migration

In `Dragonfly`, data migration between NUMA nodes is performed on a per-application basis. When applying MTP, the user-level application monitors the migration triggering values inside the kernel periodically and triggers data migration via a system call.

B. Results

We experimented with two cases (Heavy, Light) for each application scenario. In each experiment, T_{window} in MTP is set to 6 seconds. We discuss how to set an optimal T_{window} in MTP in detail later. Figure 6(a) is a case where the iMC of Node0 is overloaded. When the iMC is overloaded, `NThread` no longer proceeds with thread migration. Thus, there is no difference in throughput between baseline and `NThread`. However, `Dragonfly` migrates files that are being remotely accessed and distributes the iMC's load of Node0 to Node1, allowing all threads to perform local read I/O. As a result, `Dragonfly` shows a performance improvement of 1.5× in a Webserver application scenario, 1.2× in Webproxy, and 7.1× in Videoserver compared to `NThread`. On the other hand, in the case of the Fileserver application scenario where the write ratio is relatively high, local writes and reads are performed dependent on the location of the thread. So, all of baseline, `NThread`, and `Dragonfly` show similar results.

Figure 6(b) is a situation where there is no iMC overload in the iMC of Node0. Therefore, `Dragonfly` does not migrate data so its performance is similar to `NThread`. Overall, the throughput of `NThread` and `Dragonfly` increases 28.9% compared to baseline.

The effectiveness of MTP is determined by the value of T_{window}. The optimal value for T_{window} varies depending on the application. We describe below how to find the optimal T_{window} value for a Webserver (Heavy) application. First we find the value of K in Equation 7. This value is obtained by TH_l, TH_r, and O_m. TH_l is the application throughput when all files of this application are accessed locally. On the other hand, TH_l is the application throughput when remotely accessing all files of this application. In our experiment, these values were measured to be 20.7 GB/s and 13.5 GB/s, respectively. Next, we find the time (T_m) it takes to migrate all the files of this application from remote memory to local memory. In our experiment, this value was measured to be 3.5 seconds. Then we get the calculated values of $TH_l \times T_m$ and $TH_r \times T_m$ respectively. And the difference between

these two calculated values is O_m. In our experiment, O_m was calculated as 30.6 GB. So we can compute the value of K using TH_l, TH_r, and O_m. In our experiment, the value of K is 4.25. Finally, we can confirm that there is a performance gain by setting the value of T_{window} to 6 seconds based on Equation 9.

Figure 6(c) shows the measurement result of the application throughput by changing the value of T_{window} for the Webserver (Heavy) application. From the figure, we see that the highest throughput is when the T_{window} is 6 seconds. We also observe that if the T_{window} is larger than 6 seconds, the performance decreases due to the lower migration trigger frequency. This means that files are accumulated in the request queue, but not triggered. On the other hand, if T_{window} is smaller than 6 seconds, the performance decreases because the amount of time to benefit from local access after migration is reduced. In the red region, the migration overhead is greater than the benefit from local access gain through migration, so the throughput is even lower than the dotted red line, indicating the only remote access throughput without migration.

Finally, we used Webserver (Heavy) application and measured the application throughput by increasing its overall execution time to see the performance change according to the execution time of each application time. Figure 6(d) shows the performance of `Dragonfly` and `NThread`. We observed that `NThread` has no performance difference as runtime changes. Whereas, `Dragonfly` throughput increases as runtime increases. The reason is that the longer the runtime, the longer the time to benefit from local access.

V. Conclusion

In this paper, we found that `NThread` does not migrate threads if the integrated memory controller (iMC) of the CPU node to which the threads are migrated is overloaded, but rather, there is a performance benefit to migrating data to the location of the threads. To this end, we propose `Dragonfly`, a data migration technique for the NVM file system. `Dragonfly` goes beyond the limits of thread migration and maximizes local access through data migration. This data migration entails the effect of distributing the load of the overloaded iMC among the iMCs. Also, `Dragonfly` precisely decides whether to migrate data using a model-based data migration trigger policy (MTP) and minimizes performance overhead caused by reckless data migration in the file system. We implemented both `NThread` and `Dragonfly` in NOVA and evaluated them in an Intel Optane DC PM server running Linux kernel v5.1.0. As a representative experimental result, `Dragonfly` showed a 7.1× higher throughput than `NThread` in the Filebench's Videoserver application scenario.

VI. Acknowledgments

We thank the reviewers for their constructive comments that have improved the paper. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035, Research on High Performance and Scalable Manycore Operating System).

References

[1] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '20, 2020.

[2] A. Khan, H. Sim, S. S. Vazhkudai, J. Ma, M.-H. Oh, and Y. Kim, "Persistent memory object storage and indexing for scientific computing," in *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 1–9, 2020.

[3] B. Lepers, V. Quéma, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *Proceedings of the USENIX Annual Technical Conference*, ATC '15, 2015.

[4] J. W. Kim, J.-H. Kim, A. Khan, Y. Kim, and S. Park, "Zonfs: A storage class memory file system with memory zone partitioning on linux," in *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pp. 277–282, IEEE, 2017.

[5] T. Kim, A. Khan, Y. Kim, P. Kasu, S. Atchley, and B. B. Fraguela, "Numa-aware thread scheduling for big data transfers over terabits network infrastructure," *Sci. Program.*, vol. 2018, Jan. 2018.

[6] S.-H. Lim, J.-S. Huh, Y. Kim, G. M. Shipman, and C. R. Das, "D-Factor: A Quantitative Model of Application Slow-down in Multi-Resource Shared Systems," in *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, 2012.

[7] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "HiNUMA: NUMA-Aware Data Placement and Migration in Hybrid Memory Systems," in *Proceedings of the IEEE 37th International Conference on Computer Design*, ICCD '19, pp. 367–375, 2019.

[8] S. Yu, S. Park, and W. 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, 2017.

[9] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '16, 2016.

[10] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible File-system Interfaces to Storage-class Memory," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, pp. 14:1–14:14, 2014.

[11] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 494–508, 2019.

[12] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and Availability via Client-local NVM in a Distributed File System," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pp. 1011–1027, Nov. 2020.

[13] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A Cross Media File System," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, p. 460–477, 2017.

[14] J.-H. Kim, J. Kim, S. Park, and Y. Kim, "pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, 2019.

[15] J.-H. Kim, Y. Kim, S. Jamil, C.-G. Lee, and S. Park, "Parallelizing Shared File I/O Operations of NVM File System for Manycore Servers," *IEEE Access*, vol. 9, pp. 24570–24585, 2021.

[16] J. Wang, D. Jiang, and J. Xiong, "NUMA-Aware Thread Migration for High Performance NVMM File Systems," in *Proceedings of the 36th International Conference on Massive Storage Systems and Technology*, MSST '20, 2020.

[17] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *;login: The USENIX Magazine*, vol. 41, pp. 6–12, March 2016.