FragPicker: A New Defragmentation Tool for Modern Storage Devices

Jonggyu Park
Sungkyunkwan University
jonggyu@skku.edu

Young Ik Eom
Dept. of Electrical and Computer Engineering/
College of Computing and Informatics,
Sungkyunkwan University
yieom@skku.edu

Abstract

File fragmentation has been widely studied for several decades because it negatively influences various I/O activities. To eliminate fragmentation, most defragmentation tools migrate the entire content of files into a new area. Unfortunately, such methods inevitably generate a large amount of I/Os in the process of data migration. For this reason, the conventional tools (i) cause defragmentation to be time-consuming, (ii) significantly degrade the performance of co-running applications, and (iii) even curtail the lifetime of modern storage devices. Consequently, the current usage of defragmentation is very limited although it is necessary.

Our extensive experiments discover that, unlike HDDs, the performance degradation of modern storage devices incurred by fragmentation mainly stems from *request splitting*, where a single I/O request is split into multiple ones. With this insight, we propose a new defragmentation tool, FragPicker, to minimize the amount of I/Os induced by defragmentation, while significantly improving I/O performance. FragPicker analyzes the I/O activities of applications and migrates only those pieces of data that are crucial to the I/O performance, in order to mitigate the aforementioned problems of existing tools. Experimental results demonstrate that FragPicker efficiently reduces the amount of I/Os for defragmentation while achieving a similar level of performance improvement to the conventional defragmentation schemes.

CCS Concepts: • **Information systems** → **Storage management**; • **Software and its engineering** → **Software maintenance tools**.

Keywords: Filesystem, fragmentation, operating system, system management

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483593

ACM Reference Format:
Jonggyu Park and Young Ik Eom. 2021. FragPicker: A New Defragmentation Tool for Modern Storage Devices. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483593

1 Introduction

File fragmentation refers to the phenomenon in which a file is scattered across multiple non-contiguous areas instead of a single contiguous one. Fragmentation has been extensively studied for many years because of its negative influence on I/O performance [24–26, 30, 37, 38, 41–43, 50, 51, 55, 56, 62]. Previously, fragmentation had been considered harmful only to HDDs (Hard Disk Drives) but not to SSDs (Solid State Drives), because SSDs do not require seek time. However, recent studies [24, 25, 37, 38, 41, 50, 62] have broken the outdated myth [3, 12–14] that SSDs are immune to fragmentation, via in-depth analyses on real-world fragmentation with the representative filesystems.

To eliminate existing fragmentation, some of the current mainline filesystems provide their own defragmentation tools [2, 5, 8] such as `e4defrag` [8]. Unfortunately, there are two main problems with the existing defragmentation tools. First, such tools are filesystem-dependent. Defragmentation requires migration of existing file data blocks, and each filesystem adopts its own block management scheme. For example, while Ext4 [47] supports in-place updates, Btrfs [53] adopts an out-place update policy, which allocates new blocks for every update. Therefore, each defragmentation tool is implemented for its specific filesystem, thereby becoming filesystem-dependent.

Second, most of the defragmentation tools [2, 6–8] rely on migration of the entire content of each fragmented file into new space, which generates excessive I/O operations. The I/O operations incurred by defragmentation contribute to its time-consuming attribute and significantly degrade the performance of co-running applications. Additionally, such methods significantly increase the write traffic to the underlying storage, and thus curtail the lifetime of modern storage devices, including flash-based storage devices and even Optane SSDs. Flash-based storage supports a finite number of program/erase cycles [34, 36, 39, 46, 54, 61].

When the number of cycles reaches the threshold, the device becomes unavailable, and thus we cannot write new data into it. Optane SSDs, the state-of-the-art storage device, provide comparatively better endurance; Intel Optane SSD 905P provides 10 DWPD (Drive Write Per Day) with a 5-year limited warranty [15]. However, in the big data era, data size has been exponentially growing [48, 52], which can even threaten the lifetime of Optane SSDs. To prolong the lifetime of such devices, it is necessary to minimize the write traffic to the devices. However, conventional defragmentation schemes generate a significant amount of bulk writes which are harmful to the device lifespan. For this reason, many storage experts recommend against performing defragmentation frequently on modern storage devices, if ever [3, 12–14].

In this paper, we first investigate the characteristics of performance degradation induced by fragmentation, in a statistical way, while considering the internals of storage devices. Fragmentation affects storage devices differently depending on the internal mechanism by which each device retrieves data. For example, an HDD retrieves data by moving its head to the physical location of data, which is called seek. Therefore, fragmentation increases the seek time, thereby degrading the performance of HDDs. On the other hand, modern storage devices such as Flash SSDs and Optane SSDs do not entail such seek time. Therefore, the distance between file blocks is mostly irrelevant to the performance of such devices.

However, modern storage devices still suffer from fragmentation due to the current design of the I/O stack. In the current I/O subsystem, I/O-related structures can represent only a contiguous area in terms of LBA (Logical Block Address). Therefore, when an application issues a system call towards data that consist of several separate filesystem blocks, the system call is divided into multiple small random I/O requests. This phenomenon i) exacerbates the kernel overheads for generating and managing I/Os, ii) increases the interface overheads between the host and the device, and iii) hinders efficient utilization of device resources.

Based on the aforementioned insights about fragmentation, we propose a new filesystem-agnostic defragmentation tool for modern storage devices, called FragPicker. The main goal of FragPicker is to minimize the amount of I/Os induced by defragmentation while achieving a similar level of performance improvement as conventional tools that fully migrate each file. To achieve this, FragPicker first analyzes the I/O characteristics of applications and finds out the optimal set of data in each file for migration. Afterwards, FragPicker migrates the fragmented data into a new space without utilizing filesystem-specific functions.

To verify the efficacy of FragPicker, we conducted extensive evaluation with various benchmarks, including both synthetic and macro benchmarks. In our experiments, FragPicker shows a similar level of performance improvement to the existing defragmenters where the files are fully migrated, such as `e4defrag`, while greatly reducing the write traffic. In particular, with YCSB workload-C running on an Optane SSD, FragPicker decreases the elapsed time of defragmentation to 16% of `e4defrag` while reducing the performance degradation of co-running applications by around 45%. Additionally, the amount of writes is reduced by around 66% with 3.4% difference in performance, compared with `e4defrag`.

Overall, this paper makes the following contributions:

- Statistical analysis of performance degradation induced by fragmentation, considering internals of storage devices. (Section 3)
- Analyzing application I/Os and clarifying the optimal pieces of data for migration without any kernel modification. (Section 4.1)
- Migration of data without utilizing any filesystem internal functions for filesystem independence. (Section 4.2)

2 Background

2.1 File Fragmentation

File fragmentation is a phenomenon in which a file consists of multiple non-contiguous blocks. Due to its negative influence on I/O performance, each filesystem tries to prevent fragmentation by its own policies. For example, the Ext4 filesystem delays block allocation until data are flushed from the page cache, and assigns multiple blocks at a time to prevent fragmentation [44]. Despite such efforts, it has been reported that fragmentation still occurs after repetitive I/O activities regardless of the filesystem type [24, 25, 30, 38, 41]. Specifically, [30] observes that 14%-33% of files are fragmented on their inspected smartphones. [38] shows the average size of fragment in newsfeeddb-journal of Facebook is only 7KB on one-year-old Nexus5. Finally, [24, 25, 41] demonstrate that modern filesystems do age, often severely, even on SSDs that have enough free space, under realistic workloads including databases and mail-server, despite existing optimizations for preventing fragmentation.

2.2 Implications of Fragmentation on Performance

Fragmentation hinders I/O activities mainly because of two aspects. First, the file blocks of fragmented files disperse over a larger space than those of non-fragmented files. Therefore, accessing fragmented files demands longer seek time on HDDs. Second, as depicted in Figure 1, since a single I/O request in the Linux I/O stack can express only a contiguous LBA area, fragmentation divides a single I/O request into multiple small ones, which increases the number of I/O requests and their randomness. Moreover, in the case of synchronous I/Os, all the requests derived from a system call should be completed to resume the caller process, thereby degrading I/O performance. We refer to this as *request splitting* in this paper.

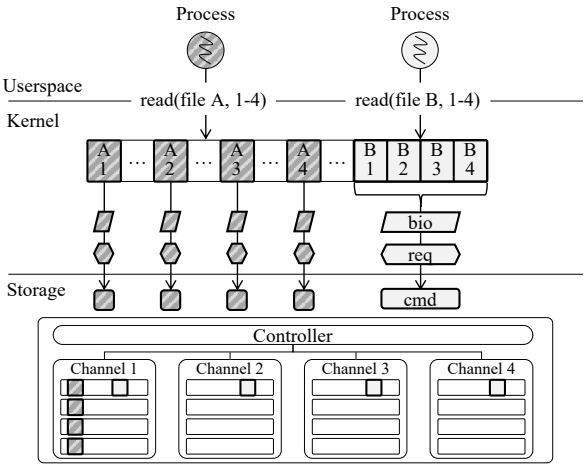


Figure 1. The fragmentation problem and the parallel architecture of SSDs

Previously, modern storage devices (Flash/Optane storage) had been considered resistant to fragmentation because retrieving data on such devices does not involve seek time, unlike HDDs. However, recent studies have revealed that even modern storage devices experience performance degradation due to fragmentation [24, 38, 41, 50, 62]. In modern storage devices, *request splitting* incurs the following problems.

First, an increase in the number of I/O requests aggravates **the kernel overheads** for creating and managing I/Os. As shown in Figure 1, the read system call to file-B can be handled by only a single request, whereas the one to file-A requires four different I/O requests. Therefore, the Linux kernel needs to create four pieces of bio structure, request structure, and I/O command. Additionally, since the I/O schedulers manage the I/O requests in the block layer for fairness and performance, *request splitting* increases the management overheads for the I/O requests. Second, **the interface overhead** between the host and the storage devices can increase due to a rise in the number of I/O commands. This interface overhead becomes severe, especially on MicroSD and eMMC devices that do not support command queuing. Since such devices can accept only a single I/O command at a time [50, 57], *request splitting* exacerbates this I/O serialization, resulting in enormous performance degradation.

Finally, *request splitting* hinders efficient utilization of internal resources inside storage devices. Modern storage devices such as SSDs consist of multiple internal units that can be accessed in parallel [22, 23, 27, 28, 32, 35, 40, 59]. The parallel architecture enables such devices to maximize I/O performance by simultaneously accessing data from multiple internal units. However, fragmentation decreases the size of I/O requests and increases their randomness, which prevents the parallel units from being fully utilized [23, 40]. Specifically, if the size of I/O requests from the host is smaller than

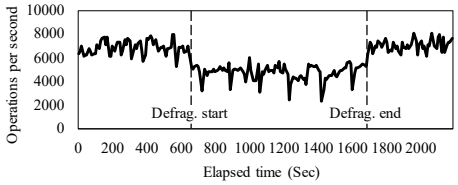


Figure 2. The performance of YCSB workload-A with background defragmentation

the maximum size that an SSD can fetch at a time with its parallelism, the SSD cannot exploit all the parallel units for the I/O requests. Additionally, the increased randomness of I/Os precipitates **resource conflicts** such as channel conflicts, since the corresponding data can exist on the same channel, as illustrated in Figure 1. Finally, an increase in the number of I/O commands often overloads the in-storage CPU cores for processing I/O commands, especially on low-end storage devices.

Although many previous studies [24, 30, 50] have focused on the read performance regarding fragmentation, fragmentation can even impede various I/O activities besides read operations. While flash-based storage does not allow in-place updates, Optane SSDs perform in-place updates. In other words, flash-based storage invalidates the existing data and sequentially allocates new flash pages across multiple channels [33]. For this reason, the internal resource conflicts induced by fragmentation can be avoided on the flash-based storage devices in the case of update operations. On the other hand, Optane SSDs reuse the original pages and write the updated data in place [29, 31, 59, 60]. Thus, even update performance can be degraded by fragmentation on Optane SSDs for the same reason as the read performance. Besides, a discard command can also represent only a contiguous LBA area. Therefore, fragmentation increases the number of discard commands for the same amount of data, thereby raising the discard cost.

2.3 Conventional Defragmentation Tools

Each filesystem usually has its own defragmentation tool that eliminates existing fragmentation. For example, the Ext4 filesystem provides `e4defrag` [8], which eliminates fragmentation of extent-based files. The `e4defrag` migrates the entire contents of a target file to a new contiguous area without considering the performance characteristics of modern storage devices. In this process, the defragmentation tool reads the entire data and writes them into new blocks, which incurs excessive I/O operations. Such additional I/O operations significantly prolong the elapsed time of defragmentation and tremendously degrade I/O performance of co-running applications. Moreover, the excessive bulk writes of defragmentation accelerates the wear-out of modern storage devices. The F2FS filesystem [45] also provides a defragmentation tool, called `defrag.f2fs` [5]. Similarly to `e4defrag`, it shows a

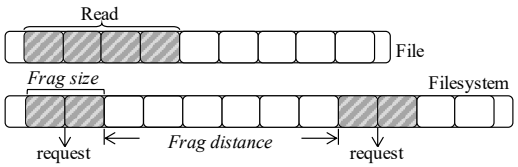


Figure 3. Fragmentation metrics

lack of ability to minimize the amount of I/Os during defragmentation although F2FS is designed for flash-based storage. Additionally, the interfaces of `defrag.f2fs` are too difficult for novices to use because the input parameters include block-level addresses.

Finally, `btrfs.defragment` [2], the defragmentation tool of Btrfs, copies the entire file contents into a new area, similarly to other tools. However, it also provides a special scheme that ignores any fragments that are bigger than a specific size. Unfortunately, this scheme is not designed to avoid *request splitting*, so a misuse of this scheme can result in misalignment of fragments with I/O requests. For instance, even if the size of a fragment equals the I/O request size, a request can be split when the corresponding data lie on two different fragments. Therefore, it can still cause *request splitting* even after defragmentation. Additionally, the default fragment size of this scheme is 32MB, and Btrfs advises that the reasonable values are from tens to hundreds of megabytes [2], which are much larger than the usual request size.

2.4 Overheads of Running Defragmentation tools

To quantitatively measure the overheads of defragmentation, we ran workload-A of YCSB with RocksDB on Flash SSD while defragmenting a thousand of unrelated files (30GB in total) using `e4defrag`. As shown in Figure 2, the performance (operations per second) of workload-A starts to decline as `e4defrag` begins due to I/O operations generated by defragmentation. Specifically, `e4defrag` degrades the performance of YCSB by around 32% on average because it issues around 30GB of both read and write operations. Additionally, the defragmentation took over 17 minutes for migrating 30GB data even with the Flash SSD.

Fragmentation is a recurring problem with a short recurrence interval (e.g., [30] explores that fragmentation recurs within a week after full filesystem-level defragmentation on their devices with Ext4). For this reason, defragmentation tools, including Microsoft drive optimizer [19] and defraggler [6], provide scheduled defragmentation, which operates daily, weekly, or monthly. Diskeeper [20] even recommends performing defragmentation daily in the case of database and email servers. Since such defragmentation targets entire filesystems or directories in practice, the aforementioned overhead of defragmentation becomes exacerbated and not trivial when it becomes a routine.

In summary, the existing defragmentation tools generate a large amount of I/Os during data migration, which make defragmentation time-consuming and significantly degrade I/O performance of co-running applications. Additionally, the excessive bulk writes of data migration curtails the lifetime of modern storage devices. Moreover, since filesystems have their own block management policies, such as CoW of Btrfs, defragmentation tools utilize filesystem internal functions, thereby becoming filesystem-dependent.

3 An Experimental Performance Analysis on Fragmentation

As mentioned before, fragmentation diminishes I/O performance regardless of the storage device types. However, due to the different internal mechanisms of storage devices, modern storage devices show disparate characteristics of performance degradation, compared with HDDs. In this section, we investigate the characteristics of performance degradation and clarify the rationale with an understanding of the internal architectures of storage devices. To achieve this, we introduce two metrics in this paper, *frag size* and *frag distance*. *Frag size* denotes the size of a fragment, and *frag distance* indicates the distance between two adjacent fragments. For instance, in Figure 3, the *frag size* is 8KB, and the *frag distance* is 24KB, when the size of each block is 4KB.

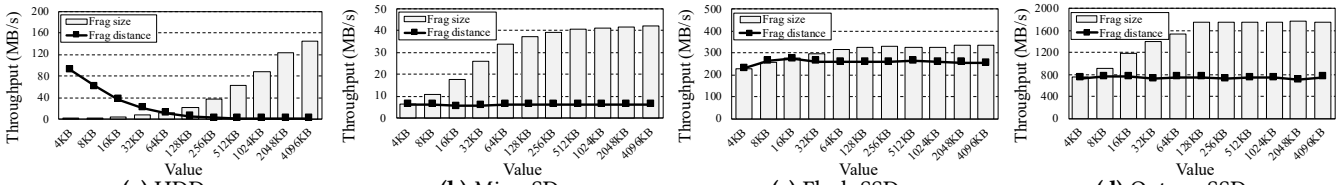
To experimentally analyze the performance degradation, we created fragmented files on F2FS [45] while varying the *frag size* and the *frag distance*, and measured the sequential read performance. Also, we calculated the correlation coefficient (CC) [21, 58] and the normalized linear regression slope (NLRS) [49] to statistically clarify the relationship between each metric and the performance. CC and NLRS are calculated by the following equations.

$$CC(X,Y) = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sqrt{\sum(x-\bar{x})^2 \sum(y-\bar{y})^2}} \quad (1)$$

$$NLRS(X,Y) = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sum(x-\bar{x})^2} \quad (2)$$

Here, X is the value of each metric, whereas Y is the normalized performance. CC indicates how related the metric is to the performance, and NLRS denotes how significantly the metric influences the performance. Here, we normalize the performance to the lowest one because the storage devices show an immense performance difference. The detailed experimental setup is described in Section 5.1.

Frag size of Figure 4 shows the sequential read performance while varying the *frag size* and fixing the *frag distance* to 1024KB. In the case of *frag distance* in Figure 4, we changed the *frag distance* while setting the *frag size* to 4KB. In all experiments, we defined the size of read requests as 128KB because it is the default readahead size in the Linux kernel.



(a) HDD **(b)** MicroSD **(c)** Flash SSD **(d)** Optane SSD

Figure 4. The sequential read performance with various frag size and frag distance

Table 1. CC and NLRS of each metric and the sequential read performance on various storage devices

Metric	Frag Size				Frag Distance	
Measure	Correlation Coefficient (CC)		Normalized Linear Regression Slope (NLRS)		Correlation Coefficient (CC)	Normalized Linear Regression Slope (NLRS)
	Before 128KB	After 128KB	Before 128KB	After 128KB		
HDD	0.9898939	0.9219008	0.1027717	0.0204404	-0.5333534	-0.0345131
MicroSD	0.8889961	0.7488593	0.0380474	0.0001394	0.2562098	0.0000272
Flash SSD	0.8437115	0.7475056	0.0027932	0.0000129	0.3109179	0.0000581
Optane SSD	0.8883913	0.4166667	0.0096721	0.0000022	-0.1539585	-0.0000091

3.1 Hard Disk Drive

As shown in Figure 4a, the sequential read performance on the HDD gradually increases as the *frag size* becomes larger. This tendency continues, even in the case when the *frag size* is bigger than the request size whereby none of the requests are split. An increase in the *frag size* can place the file on a smaller range of the storage, because we did not modify the *frag distance* and the file size here. Therefore, it can reduce the seek time and improve the sequential read performance. Likewise, the performance on the HDD tends to decrease while the *frag distance* increases. This result stems from the fact that increasing the *frag distance* extends the seek time, thereby decreasing the performance. Statistically, its CC between the *frag size* and the performance is over 0.9, and the NLRS is around 0.1 and 0.02 for the case before 128KB and after 128KB, respectively. Also, the CC between the *frag distance* and the performance is around -0.53, and the NLRS is around -0.03. According to the experiments, **the seek time is a decisive factor in the performance of HDDs.** Therefore, defragmentation tools for HDDs should focus on how to minimize the seek time.

3.2 MicroSD

On the other hand, the performance of modern storage devices exhibits a different tendency regarding fragmentation. In the case of the MicroSD card, as shown in Figure 4b, the performance gradually increases until the *frag size* becomes 128KB, while showing around 0.89 of CC and 0.03 of NLRS. This NLRS is higher than those of SSDs due to the lack of command queuing. However, when the *frag size* is larger than 128KB (the request size), the extent of the increase declines significantly, and it shows around 0.00013 of NLRS. In such cases, I/O requests are no longer divided, and thus an increase in the *frag size* does not lead to much performance improvement. Note that the MicroSD card can still achieve a minor performance gain even after the *frag size* becomes larger than 128KB because such low-end devices suffer from insufficient memory space and thus employ demand-based mapping cache. Namely, the larger *frag size* contributes to better mapping cache hit. Meanwhile, the NLRS between the *frag distance* and the performance is around 0.0000272, confirming the *frag distance* is irrelevant to the performance. Since MicroSD cards do not involve seek time unlike HDDs, **minimizing the *frag distance* is meaningless in the performance perspective.**

3.3 Solid State Drives

The Flash SSD shows a similar performance pattern to that of the MicroSD card. As shown in Figure 4c, when the *frag size* is smaller than 128KB, the CC between the *frag size* and the performance is around 0.84, and the NLRS is around 0.0027. This result comes from the fact that fragmentation increases the number of I/O requests and their randomness by splitting a single large I/O into multiple small ones [50]. As the *frag size* becomes smaller, such a phenomenon becomes more severe, thereby hindering the full utilization of internal parallelism. However, the NLRS becomes smaller than 0.00002 when the *frag size* is larger than 128KB because none of the requests are split. The NLRS between the *frag distance* and the performance is only around 0.0000581 because Flash SSDs do not embody seek time. Finally, the Optane SSD, the state-of-the-art storage device, also shows a similar performance tendency as other modern storage devices. As shown in Figure 4d, the performance of the Optane SSD is sensitive to the *frag size* until the *frag size* reaches the request size (128KB) because of the same reason as the Flash SSD. Also, on the Optane SSD, the *frag distance* is irrelevant to the performance due to the lack of seek time. Note that the NLRS of the Optane SSD is larger than that of the Flash SSD because the kernel overheads of *request splitting* is more noticeable on Optane SSDs due to their superior performance. In addition, Flash SSDs are usually equipped with a copious level of parallelism, compared with Optane SSDs [59], thereby decreasing the likelihood of internal resource conflicts.

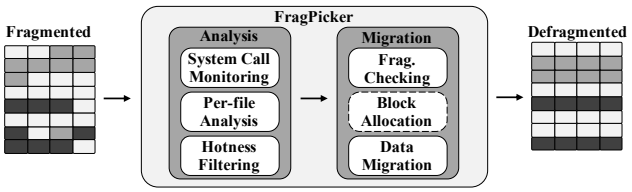


Figure 5. The overview of FragPicker

We also performed the same experiments utilizing sequential updates with O_DIRECT. Although the experimental results are not included in the form of figures due to the limited space of the paper, the performance variation of sequential updates exhibits similar tendencies as those of sequential reads for the same reason. For instance, on the Optane SSD, the CC between the *frag size* and the performance is around 0.83, and the NLRS is approximately 0.0072 when the *frag size* is smaller than 128KB. In the case of the Flash SSD, the NLRS between the *frag size* and the performance is around 0.0016, which is smaller than that with read operations. As mentioned in Section 2.1, Flash SSDs internally adopt out-place updates, unlike Optane SSDs. Therefore, update operations toward fragmented files do not reuse the original internal pages, and rather allocate new pages across multiple channels, thereby avoiding internal resource conflicts. As a result, the performance degradation of sequential updates is smaller than that of sequential reads on Flash SSDs.

Thus far, we have experimentally analyzed the performance tendencies with respect to fragmentation on various storage devices. Through the aforementioned experiments, we have discovered that **the performance of modern storage devices, such as Flash/Optane SSDs, mostly depends on the occurrence of *request splitting* and is irrelevant to the distance between fragments.** Therefore, on modern storage devices, it is more important to keep the *frag size* bigger than the request size for the sake of preventing *request splitting*, rather than to keep the *frag distance* small. This observation motivates us to design a new defragmentation tool for modern storage devices.

4 FragPicker

To alleviate the drawbacks of conventional defragmentation tools while also achieving a high level of performance, we propose a new defragmentation tool for modern storage devices, called FragPicker. The design goal of FragPicker is to i) decrease the amount of I/Os of defragmentation while achieving a similar level of performance as conventional tools, ii) become filesystem-agnostic, and iii) exclude kernel modification. To achieve this, FragPicker prevents *request splitting* without considering the *frag distance*, by leveraging the aforementioned observations that the performance of modern storage devices is unrelated to *frag distance*. FragPicker consists of two phases as described in Figure 5; **analysis** and **migration**.

In the **analysis** phase, FragPicker traces the I/O activities of applications and identifies optimal filesystem blocks for subsequent migration. Since conventional solutions migrate the entire contents of files for defragmentation, it inevitably includes meaningless copy operations. For example, when an application issues I/Os in a skewed way (non-uniform distribution), the placement and location of some hot data might decide the I/O performance. In this case, migrating cold data does not contribute to a large performance gain. To prevent such a case, FragPicker keeps track of I/O information on the system call layer and identifies the data that the application indeed requires. To further minimize the amount of I/Os, FragPicker calculates the hotness of data and utilizes it in deciding the data for migration.

In the **migration** phase, FragPicker first confirms if the pre-determined blocks in the **analysis** phase are fragmented or not. Since conventional solutions do not consider the sequentiality of blocks during migration, they migrate even contiguous pieces of data, which incurs pointless copy operations. To prevent this, before migration, FragPicker investigates the fragmentation state of such data to determine if the data needs to be migrated or not. Afterward, FragPicker performs the migration of fragmented blocks. However, each filesystem requires a different mechanism to guarantee data migration. For example, out-place update filesystems, such as F2FS and Btrfs, allocate new blocks for update operations. Accordingly, simply rewriting of data on the same file offset usually guarantees data migration. On the other hand, in-place update filesystems, such as Ext4, do not assign new blocks to updated data. Instead, they reuse the original filesystem blocks via in-place updates. Thus, FragPicker should adopt different migration mechanisms depending on the filesystems. In the case of out-place update filesystems, FragPicker simply rewrites the target data on the same file offset to migrate fragmented data. Whereas, on in-place update filesystems, FragPicker deallocates the existing blocks and allocates new blocks for the data that will be migrated, using `fallocate`, to guarantee their contiguity. By doing so, FragPicker does not utilize any filesystem specific functions unlike conventional defragmenters, thereby becoming filesystem-independent. Thus, FragPicker can operate on major types of filesystems, including Ext4, F2FS, and Btrfs.

4.1 The Analysis Phase of FragPicker

To minimize the amount of I/Os induced by defragmentation, it is important to identify the optimal set of filesystem blocks for migration. The inadequate identification can generate meaningless copy operations without any performance gain. Therefore, FragPicker performs the **analysis** phase with consideration on the I/O characteristics of applications, in order to determine the optimal set of filesystem blocks for migration. The **analysis** phase of FragPicker includes system call monitoring, per-file analysis, and hotness filtering, as illustrated in Figure 6.

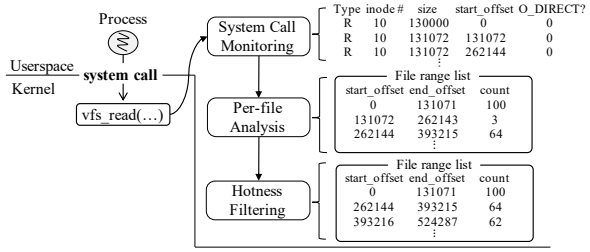


Figure 6. The analysis phase of FragPicker

4.1.1 System Call Monitoring. To analyze the I/O activities of applications, FragPicker monitors I/O-related system calls such as `vfs_read()`, `vfs_write()`, etc. For example, the parameters of `vfs_read()` contain crucial I/O information such as file structure, file offset, etc. From such system calls, FragPicker extracts the I/O type, the inode number, the I/O size, and the start offset of the I/O. Additionally, it checks whether the I/O is O_DIRECT or not . Using the inode number, we can identify which file an I/O request is trying to access. Filesystems manage a unique inode number per each file. Accordingly, FragPicker can find out the corresponding files with the inode numbers. The start offset and the I/O size are used to determine the range of file blocks that the I/O tries to access. However, in the case of buffered reads, the virtual filesystem (VFS) in the Linux kernel sometimes manipulates the range according to its read-ahead policy. Since FragPicker monitors I/O activities in the system call layer which is above the VFS, the read-ahead mechanism of the operating system cannot be applied. Accordingly, FragPicker checks the O_DIRECT flag of the system calls and imitates the read-ahead mechanism in the per-file analysis step when the I/Os are buffered sequential reads. To monitor system calls without any kernel modification, we utilize the existing profiling tool, BCC (BPF Compiler Collection) [17], which makes use of extended BPF (Berkeley Packet Filters). BCC facilitates us to extract information on a specific system call of a particular application.

4.1.2 Per-file Analysis. After monitoring I/O system calls, FragPicker analyzes the monitored I/O activities for each file and creates a file range list for each file. First, FragPicker identifies the file pathname using the inode number and collects all the I/O activities towards the file. Second, it calculates the end offset by using the start offset and the I/O size in order to represent the range of blocks requested. Here, FragPicker adjusts the start/end offsets so that the range becomes aligned with the underlying filesystem blocks. This is necessary for the prevention of data loss, which will be addressed in detail in Section 4.2.2. Besides, FragPicker imitates the behavior of the read-ahead mechanism to increase the accuracy. When the I/Os are detected as buffered sequential reads by comparing the start offset and end offset, FragPicker increases the range to the read-ahead size (by default, 128KB). FragPicker ignores the subsequent sequential reads that lie within the read-ahead window because those I/Os are serviced by the page cache.

Algorithm 1: Merging Overlapped I/Os

```
EO is end_offset
OW is overlap_window
forall entry ∈ file range list do
    if is_overlapped(entry, OW) then
        count++
        if entry.EO > OW.EO then
            OW.EO = entry.EO
    else
        store(OW, count)
        OW = entry
```

Afterward, regardless of the I/O patterns, FragPicker merges overlapped I/Os and calculates the I/O counts to reflect the hotness of the file range. FragPicker sorts the file range list based on the start offset and performs a merge process, described in Algorithm 1. Here, the overlap_window is the current representative I/O range that absorbs all the overlapped I/Os. In the merge process, FragPicker inspects whether or not each entry is overlapped with the overlap_window. If so, it increments the count and adjusts the overlap_window to merge the overlapped I/Os. If a new entry is not overlapped, FragPicker stores a pair of overlap_window and the count into the file range list, while designating the new entry as a new overlap_window.

For example, suppose that two I/Os access file ranges 1-40 and 31-60, respectively. Since the I/Os' ranges overlap on 31-40, FragPicker merges them and creates a new one which represents a range 1-60 while incrementing the I/O count. If the two I/Os were not merged and separately treated, FragPicker would migrate the corresponding data blocks independently and consequently generate fragmentation between the two file ranges. This merge process also helps decrease the amount of I/Os during the **migration** phase.

4.1.3 Hotness Filtering. To further optimize FragPicker, we implemented hotness filtering, which extracts only the hot regions. When applications issue I/O system calls in a non-uniform way, certain regions of the file will be accessed more frequently than others. In this case, eliminating fragmentation of the cold regions does not necessarily lead to performance improvement. Therefore, FragPicker sorts the range entries based on the I/O counts and truncates the cold entries. Here, it is the system administrator's decision to decide how many entries will be truncated. Since it depends on the system requirements, FragPicker leaves this as a tunable option. The smaller entries FragPicker truncates, the higher performance the system can achieve while generating a larger amount of writes.

4.1.4 Bypass Option for Avoiding Redundant Analysis. In some cases, the system administrator already comprehends the detailed I/O characteristics. For example, if

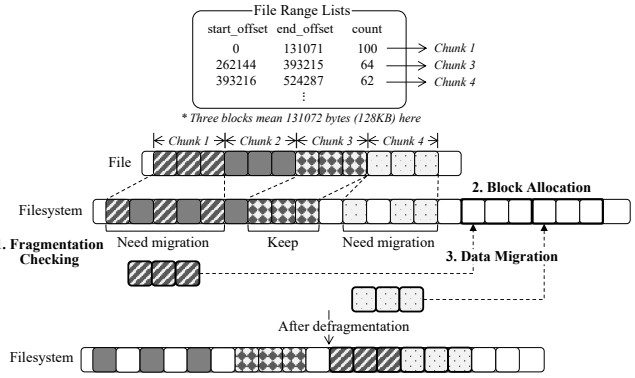


Figure 7. The migration phase of FragPicker

an application usually looks up the entire data from the beginning of a file, we can expect the application will issue sequential reads. In this case, the **analysis** phase may be redundant. For such cases, we also provide a bypass option for sequential reads, which bypasses the **analysis** phase and simply divides the file range according to the read-ahead size. In the case of sequential reads, the size of sequential read I/Os is adjusted to the read-ahead size by the inherent read-ahead mechanism. Therefore, FragPicker creates the file range lists with read-ahead sized range entries from the beginning offset. We believe this option is useful because sequential read performance is highly influenced by fragmentation. Useful instances of this option include database lookup, searching words in a directory like grep, etc.

4.2 The Migration Phase of FragPicker

The **migration** phase of FragPicker consists of fragmentation checking, block allocation, and data migration, as shown in Figure 7. First, fragmentation checking confirms if the data on each file range in the file range list are indeed fragmented on the filesystem layer. If the data are not fragmented, they do not need to be migrated. Second, FragPicker pre-allocates contiguous blocks so that the following data migration can place the data into a contiguous area. Finally, FragPicker migrates the data into the new area and eliminates fragmentation.

4.2.1 Fragmentation Checking. The **analysis** phase of FragPicker determines the optimal pieces of data for migration considering I/O activities and their hotness, and records the corresponding file range in the file range lists. However, if the piece of data is not fragmented, migration of the data does not enhance performance and rather induces wasteful copy operations. To further reduce the amount of I/Os incurred by data migration, FragPicker checks whether the data on each file range are fragmented or not. To collect such fragmentation information, we utilize the existing `filefrag` program [10] which reports on the file fragmentation information using FIEMAP ioctl. The `filefrag` program provides mapping information between file offsets and logical block addresses. Utilizing `filefrag`, FragPicker obtains the corresponding LBAs for file offsets and examines the sequentiality of the LBAs of the data on each file range. If any of the LBAs are not sequential, FragPicker determines that the data are fragmented. Otherwise, FragPicker ignores the range entry and moves on to the next entry. Since `filefrag` is supported by most Linux filesystems, we believe that this does not degrade the versatility of FragPicker.

For instance, in Figure 7, chunk 3 is not fragmented on the filesystem layer, and thus a single I/O request can still represent the data without *request splitting*. Therefore, FragPicker excludes chunk 3 in migration, regardless of the pre-performed analysis. Also, FragPicker ignores chunk 2 even though the chunk is fragmented, because the **analysis** phase determined that chunk 2 was not frequently accessed by the applications. Finally, chunk 1 and chunk 4 are fragmented and frequently accessed by applications. Therefore, FragPicker performs block allocation and data migration only for these chunks.

4.2.2 Block Allocation. As previously mentioned, each filesystem has its own internal mechanism of block management. On out-place update filesystems such as F2FS and Btrfs, simply overwriting data guarantees migration into a new area. However, in-place update filesystems, such as Ext4, reuse the existing blocks for updates. Therefore, FragPicker should allocate a contiguous area in advance for the subsequent migration, so as to eliminate fragmentation. We utilize the `fallocate` system call [9] to solve this problem without using filesystem specific functions. The `fallocate` system call allows partial control on the block management from the userspace. Using `fallocate`, we can allocate/deallocate filesystem blocks for a certain range of a file. FragPicker first deallocates the file data blocks in the range that needs to be migrated, using the start/end offsets. Afterward, FragPicker allocates a new contiguous area for the deallocated data so that the subsequent migration can place the data conjointly. In the meantime, FragPicker temporarily stores the corresponding data in the internal buffer. The entire process of this block allocation is protected by file locking to prevent concurrent access. FragPicker skips this block allocation for out-place update filesystems because rewriting data on the same offset guarantees data movement in such filesystems even without this block allocation step.

One might wonder whether the deallocation process induces potential data loss upon sudden power off. Deallocation of `fallocate` performs zeroing data that are not aligned with the filesystem blocks, to prevent falsely reading the deallocated data. However, since the per-file analysis step already adjusted the file ranges to align them with the filesystem blocks, deallocation in FragPicker does not zero any of the existing data. Additionally, in the case of Ext4, block deallocation is protected by journaling to maintain filesystem-consistency. It means the metadata (inode, bitmaps, etc.)

Table 2. Experimental setup

CPU	Intel Xeon E5-2620 v4 2.1GHz 8 core
Memory	DDR4 2133MHz 16GB
Storage	Samsung HDD 7200RPM 1TB Samsung MicroSD card EVO type A1 128GB Samsung SATA Flash SSD 850 PRO 256GB Intel NVMe Optane SSD 905P 960GB
OS	Ubuntu 18.04 LTS with Linux Kernel 5.7.0

related to deallocation are not immediately reflected into the storage and wait for the journal transaction to be committed. Meanwhile, the filesystem prohibits reuse of the deallocated blocks until the corresponding transaction is committed, to guarantee filesystem-consistency [18]. In this circumstance, we believe the data loss due to block zeroing or reuse of the deallocated blocks does not occur in practice. Thus, since we do not delete the file range lists before guaranteeing the success of defragmentation, we can recover the deallocated data even after sudden power off, using debugfs [4], the file range lists, and the mapping information.

4.2.3 Data Migration. After the previous steps, FragPicker is ready to migrate the fragmented data. It migrates the fragmented data by rewriting the data on the same offset. As shown in Figure 7, the fragmented data on chunk 1 and 4 are migrated into a new area, and they become contiguous. Therefore, we can efficiently eliminate fragmentation so that the subsequent I/O requests towards the file are no longer split. Since FragPicker migrates only partial data rather than the entire files, it can effectively minimize the amount of I/Os during defragmentation. The order of chunks of a file may not be sequential, and the distance between each chunk can increase. For example, after defragmentation, the order of the chunks in Figure 7 becomes non-sequential, and the distance between each chunk increases. However, the order and distance of the chunks are not the decisive factors in the performance of modern storage devices, as shown in the motivational experiments of Section 3. Meanwhile, although some cold chunks, like chunk 2, are still fragmented, such chunks are not critical to the performance according to the result of the **analysis** phase. Therefore, FragPicker can achieve a similar level of performance improvement to the case where the entire file is fully migrated.

5 Evaluation

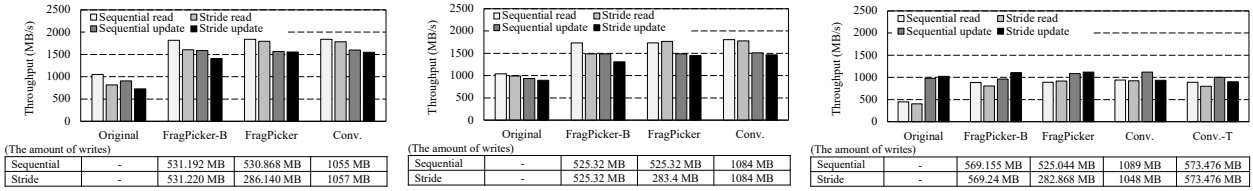
5.1 Experimental Setup

We implemented FragPicker fully on the userspace without any kernel modification and utilization of filesystem specific functions. F2FS sometimes performs in-place updates to minimize the segment cleaning overheads. Therefore, FragPicker temporarily disables in-place updates via `sysfs` right before the migration and restores the previous state after the migration. For the conventional defragmenters (Conv. in Figure 8, 9, 11), we utilize `e4defrag` for Ext4 and `btrfs.defragment` for Btrfs. Note that since F2FS does not provide user-friendly defragmentation tools that defragment data in a unit of file or directory, we instead defragmented files by mimicking the behavior of other defragmentation tools. To evaluate FragPicker, we ran synthetic, database, and fileserver benchmarks on a system equipped with various types of storage devices, as described in Table 2. In the case of FragPicker, we performed the workload analysis prior to migration by running the same workloads except for the YCSB experiment.

5.2 Synthetic Workloads

For synthetic workloads, we artificially created a fragmented file that comprises a series of thirty-two 4KB blocks and one 128KB block. To create such files, on F2FS and Btrfs, we repeatedly wrote thirty-two 4KB blocks and one 128KB block in a row with O_DIRECT while simultaneously creating a dummy file in an interleaving way. On Ext4, we additionally used `fallocate` to place filesystem blocks in a similar way to other filesystems. In this experiment, we measured the performance of sequential reads, stride reads, sequential updates, and stride updates, so as to compare FragPicker with the existing tools. At the same time, we also measured the amount of writes incurred by defragmentation to verify the efficiency of FragPicker, using blktrace [1]. Here, we set the stride size as 288KB. Since the amount of writes for defragmentation is unrelated to the I/O type (read or updates), we present the amount of writes according to the I/O patterns (sequential or stride) in the table beneath each figure.

5.2.1 Optane SSD. Figure 8 shows the performance results (for 1GB file) on Optane SSD. Here, FragPicker-B utilizes the bypass option for sequential reads without the **analysis** phase. In the case of sequential reads on Ext4, FragPicker shows around 75% higher performance than the original which is before defragmentation. This result originates from the fact that FragPicker efficiently defragments the target file and prevents *request splitting*. The performance gap between FragPicker and `e4defrag` shows only around 0.1% while FragPicker reduces the amount of writes by approximately 50%. Due to its **analysis** phase and fragmentation checking, FragPicker can migrate only the optimal pieces of data in a file that critically influence I/O performance. FragPicker-B also shows a similar performance to FragPicker while they both show almost the same amount of writes. Since the bypass option automatically generates the file range lists for sequential reads, FragPicker-B can show the same-level of defragmentation efficiency even without the complicated **analysis** phase. However, in the case of stride reads, FragPicker shows around 12% higher throughput than FragPicker-B while FragPicker generates around 45% lower amount of writes. Unlike FragPicker-B, FragPicker performs a comprehensive analysis on the I/O activities and thus can efficiently defragment only necessary file blocks even under stride workloads. In the case of FragPicker-B, the stride read operations incur *request splitting* due to the misaligned



(The amount of writes)

Sequential	-	531.192 MB	530.868 MB	1055 MB	
Stride	-	531.220 MB	286.140 MB	1057 MB	

(a) Ext4

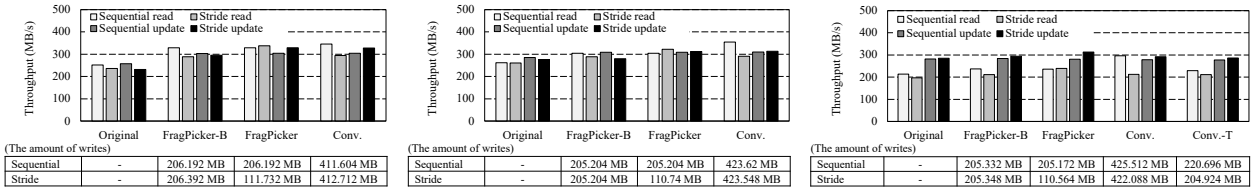
Sequential	-	525.32 MB	525.32 MB	1084 MB	
Stride	-	525.32 MB	283.4 MB	1084 MB	

(b) F2FS

Sequential	-	569.155 MB	525.044 MB	1089 MB	573.476 MB
Stride	-	569.24 MB	282.868 MB	1048 MB	573.476 MB

(c) Btrfs

Figure 8. The experimental results of the synthetic workloads on the Optane SSD (1GB file)



(The amount of writes)

Sequential	-	206.192 MB	206.192 MB	411.604 MB	
Stride	-	206.392 MB	111.732 MB	412.712 MB	

(a) Ext4

Sequential	-	205.204 MB	205.204 MB	423.62 MB	
Stride	-	205.204 MB	110.74 MB	423.548 MB	

(b) F2FS

Sequential	-	205.332 MB	205.172 MB	425.512 MB	220.696 MB
Stride	-	205.348 MB	110.564 MB	422.088 MB	204.924 MB

(c) Btrfs

Figure 9. The experimental results of the synthetic workloads on the Flash SSD (400MB file)

block layout with I/Os, thereby achieving a lower level of performance than FragPicker.

Since Ext4 performs in-place updates, the update performance is also affected by fragmentation, similarly to that of read operations. Moreover, Optane SSD also performs in-place updates internally. Therefore, fragmentation incurs *request splitting*, and in turn causes internal resource conflicts for update operations. As a result, FragPicker enhances the sequential update performance by up to 73%. Meanwhile, the performance difference between FragPicker and `e4defrag` is only 2%. Stride updates also show similar performance results as others. FragPicker achieves a similar level of performance improvement while inducing only around 27% of writes compared with `e4defrag`.

The performance results with F2FS is presented in Figure 8b. Similarly to Ext4, FragPicker with F2FS improves the performance of sequential read, sequential update, stride read and stride update by up to 67%, 60%, 79%, and 62%, respectively. Additionally, FragPicker generates 52% (Sequential) and 74% (Stride) lower amount of write operations during defragmentation, compared with the conventional scheme. Note that FragPicker can accomplish a performance gain with update workloads because F2FS performs in-place updates in the case of O_DIRECT operations.

Figure 8c shows the experimental results of Btrfs. Different from other experiments, we additionally performed an experiment using the Btrfs defragmenter with its optimization that ignores any extents bigger than a specific size. We set the size as 128KB (request size) and denote this as Conv.-T in the figure. In the case of read operations, FragPicker achieves a similar level of performance enhancement as the case where the entire file contents are migrated (Conv. in the figure) while minimizing the amount of writes. Additionally, although not shown in the figure, we measured the elapsed time of defragmentation with the stride workloads, and FragPicker decreased the time by 78% due to its efficient defragmentation. In the meantime, the stride read performance of Conv.-T is 13% lower than FragPicker while generating around 2x more writes. The misalignment of fragments with I/O requests due to the optimization generates additional *request splitting* even after defragmentation.

The update performance on Btrfs differs from that on other filesystems. The Ext4 filesystem adopts in-place updates, and F2FS sometimes performs in-place updates to avoid frequent segment cleaning. On the other hand, the default version of Btrfs performs out-place updates, and thus fragmentation does not affect the performance of update operations on Btrfs. Therefore, regardless of I/O patterns, defragmentation could not enhance the update throughput in Btrfs.

5.2.2 Flash SSD. Figure 9a shows the performance results (for 400MB file) on Ext4 with a SATA Flash SSD. As shown in the figure, FragPicker enhances the sequential read performance by up to 30%, compared with the original. The performance gain from defragmentation is smaller than that on the Optane SSD. On Flash SSDs, the kernel overheads for managing an increased number of I/O requests due to fragmentation are less noticeable than the Optane SSDs because of their higher storage latency.

In the case of stride reads, FragPicker even outperforms the conventional defragmenters. FragPicker migrates fragmented data in the way the data are accessed. Therefore, FragPicker can place the data in a better way than migrating the entire contents sequentially. Specifically, even if the entire file is placed contiguously on the filesystem layer, reading the file in a stride way can cause internal resource contention. Therefore, FragPicker can exploit the internal parallelism of the underlying Flash SSDs better than the others. Meanwhile,

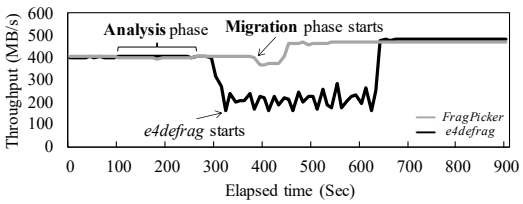


Figure 10. YCSB workload-C with RocksDB on Ext4

the performance gain of defragmentation with update operations is smaller than that with read operations because Flash SSDs allocate new pages across multiple channels for update operations. As a result, FragPicker enhances the sequential update performance by 17%.

Finally, we also measured the discard cost (s/GB) by deleting the file and issuing discard using fstrim command. Since a discard command can also represent only contiguous LBAs, fragmentation brings about a large number of discard commands. Since F2FS and Btrfs have recently decoupled the discard operations from the critical path by performing them asynchronously, we measured the discard cost only on Ext4 with the Flash SSD. As a result, FragPicker reduced the discard cost from 16.6 s/GB to 8.485 s/GB via its efficient defragmentation.

In summary, FragPicker efficiently reduces the amount of I/Os incurred by defragmentation while achieving a similar level of performance improvement as the existing tools, regardless of the types of filesystems, I/O patterns, or the storage devices. Moreover, FragPicker even outperforms the existing tools in terms of stride I/O performance on Flash SSD, possibly due to improved utilization of internal parallelism.

5.3 Database Workloads

5.3.1 RocksDB. To comprehensively evaluate the efficacy of FragPicker, we ran YCSB workload-C (Read 100% and zipfian distribution) using RocksDB with the Ext4 filesystem on the Optane SSD. We created 20,000,000 records and configured the block size of RocksDB as 128KB. Additionally, the read/write option of RocksDB is set as O_DIRECT, while all of the other options are set to the default. First, we aged the filesystem with dummy files using *Dabre* profile [41], which is captured in 2017 from the root partition of the one-year-old Ext4 filesystem. Afterward, we loaded the workload data and deleted some of the dummy files to secure some free space. Finally, we ran the workload while performing defragmentation using both `e4defrag` and FragPicker, and measured the total throughput using `iotop` [16]. We set the hotness criterion of FragPicker as the top 50% in this experiment.

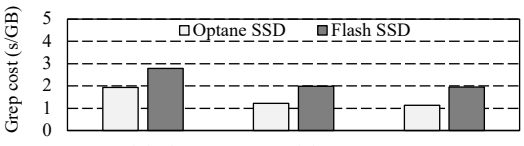
As shown in Figure 10, `e4defrag` took 331 seconds to perform defragmentation for around 23GB database files and degraded the workload throughput by around 47%. This is because `e4defrag` issued excessive I/O operations for defragmentation. Moreover, we observed that, unlike FragPicker, `e4defrag` inefficiently issues 4KB read I/Os for reading fragmented data in migration, which further exacerbates the overhead. After defragmentation, `e4defrag` improved the throughput by 20.1%, compared with that before defragmentation. In the case of FragPicker, the **analysis** and **migration** phase degraded the throughput by only 1.4% and 7.4%, respectively. Additionally, FragPicker decreased the elapsed time of defragmentation to around 54 seconds. Finally, the throughput after defragmentation of FragPicker is only 3.4% lower than that of `e4defrag`. This result originates from the fact that the **analysis** phase of FragPicker efficiently detects the performance-critical pieces of data with negligible overhead, and the **migration** phase of FragPicker migrates a minimal amount of data via the fragmentation checking process. As a result, the total I/O amount of FragPicker is around 66% lower than `e4defrag`. The throughput difference between FragPicker and `e4defrag` can be further minimized by longer analysis time and relaxing the hotness criterion.

5.3.2 SQLite. To evaluate FragPicker with Btrfs on MicroSD, we created 100,000 entries with 4KB value size, using SQLite synchronous sequential insertion. We observed that this workload generates a severe degree of fragmentation on Btrfs even without conducting filesystem aging in advance. Afterward, we executed FragPicker and `btrfs.defragment` to defragment the SQLite files while simultaneously running the FIO benchmark [11] that issues 128KB sequential write I/Os. Finally, we measured the elapsed time of a select query, which returns 30% of the entire data.

Although the evaluation results are not included in the form of figures due to the limited space of the paper, FragPicker decreased the elapsed time of the select query from 29.472 seconds to 4.427 seconds after defragmentation. The difference in the performance gain between FragPicker and `btrfs.defragment` is only 3.5%. The performance improvement of defragmentation is much higher than other storage types because the MicroSD card we used in this experiment only accepts a single I/O command at a time. Therefore, a significant increase in the number of I/Os due to fragmentation tremendously degrades I/O performance on MicroSD. Meanwhile, FragPicker issued around 163MB read and 137MB write I/Os during defragmentation while `btrfs.defragment` generated around 474MB read and 426MB write I/Os. As a result, FragPicker reduced the elapsed time of defragmentation to 30.08%, compared with `btrfs.defragment`. Moreover, the I/O throughput of FIO with FragPicker is around 2 times higher than that with `btrfs.defragment`.

5.4 Fileserver Workload

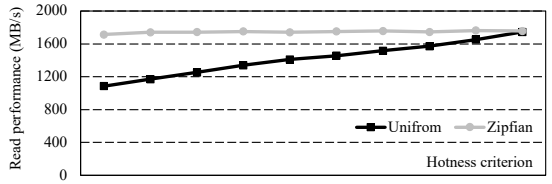
To evaluate FragPicker with F2FS, we measured the elapsed time of the recursive grep with buffered I/O after running the Filebench fileserver workloads and calculated the grep



(The amount of writes)

Optane SSD	-	27653 MB	49118 MB
Flash SSD	-	10862 MB	24597 MB

Figure 11. The Filebench fileserver workload on F2FS



(The amount of writes)

Unifrom (MB)	32	66	97	127	164	198	231	257	301	337
Zipfian (MB)	0.7	1	2.4	2.1	2.6	4.5	5.4	6.1	7.2	7.5

Figure 12. The hotness test under the Unif. and Zipf. dist.

cost via dividing the elapsed time by the entire size of the fileserver data. Since the recursive grep sequentially reads the entire files under a certain directory, the grep test is often adopted by research papers on fragmentation to measure the sequential read performance [24, 25, 50]. Here, we configured the fileserver workloads to issue O_DIRECT I/Os. We created 3000 files for the Flash SSD, and 6000 files for the Optane SSD. Here, the average file size is approximately 8.4MB. Figure 11 shows the grep cost and the amount of writes incurred by defragmentation.

Overall, FragPicker shows similar performance to the conventional scheme while greatly decreasing the amount of I/Os. In the case of the Optane SSD, the average number of fragments per file has been reduced from 1395 to 1.77 after applying FragPicker. Consequently, FragPicker shows around 37% lower grep cost than the original while showing around 44% lower amount of writes compared with the conventional scheme. On the Flash SSD, FragPicker shows around 29% lower grep cost than the original due to its efficient defragmentation. Specifically, FragPicker reduced the average number of fragments of each file from 1068 to 2.48. Note that although grep issues 32KB buffered sequential read system calls, the read-ahead detection mechanism of FragPicker detects the sequentiality of the read system calls and converts their size into 128KB. Meanwhile, the difference in the grep cost between FragPicker and the conventional scheme is only 2% on the Flash SSD.

In summary, FragPicker can achieve I/O performance improvement up to a similar level of the case where an entire file is migrated, while generating a much smaller amount of data migration. Since FragPicker migrates only partial chunks of a target file, the distance between each chunk can be further increased. However, as shown in the results, since modern storage devices do not entail seek time, it is important to prevent *request splitting*, rather than to minimize the *frag distance*.

5.5 Hotness Filtering Test

To validate the hotness filtering of FragPicker, we created the same 1GB fragmented file as the synthetic experiment on the Optane SSD, and issued 128KB read system calls with O_DIRECT in both a uniform and non-uniform way. We utilized Zipfian distribution for the non-uniform request distribution. Figure 12 shows the performance results, with relaxing the hotness criterion from 10% to 100%. The x-axis of the figure means that the top x% of hot data is migrated for defragmentation. In the case of the uniform distribution, as the hotness criterion becomes relaxed, the performance gradually increases as well as the amount of writes does. On the other hand, in the case of the Zipfian distribution, the majority of read operations access only partial data, unlike the uniform distribution. Therefore, the read performance does not noticeably change as the percentage increases. The **analysis** phase of FragPicker precisely determines the hot data, and thus, migrating only a tiny amount of data is sufficient to achieve high performance. Specifically, the amount of writes incurred by defragmentation of FragPicker is only 0.7 MB while the file size is 1GB.

In summary, the hotness filtering of FragPicker can further decrease the amount of writes during defragmentation while accomplishing a high level of performance, especially under non-uniform workloads. Also, these experimental results indicate that the higher hotness criterion results in both higher performance and a greater amount of writes, especially when applications issue non-skewed I/Os.

6 Discussion

To minimize the amount of writes during defragmentation, Hahn *et al.* [30] proposed a copyless defragmenter, called Janusd. Janusd defragments filesystems via modifying the mapping table inside the SSD instead of copying the physical data. However, Janusd is not applicable to commercial SSDs because it requires a specialized remapping operation, which is not publicly available. Additionally, Janusd is still a file-based defragmenter, which defragments the entire content of files, while FragPicker migrates only performance-critical data. Park *et al.* [50] suggested a new log-structured filesystem (LFS), called AALFS, to perform defragmentation without generating additional write operations. AALFS utilizes the segment cleaning of LFSs to relocate fragmented blocks into a contiguous area. Therefore, this scheme is only applicable to LFSs.

Some previous studies [24, 25] address both inter- and intra-file fragmentation. However, since the distance between file blocks is not critical to I/O performance on modern storage devices, inter-file fragmentation on such devices does not incur noticeable performance degradation, unlike HDDs.

Therefore, FragPicker addresses only intra-file fragmentation without considering inter-file fragmentation.

Meanwhile, internal operations of Flash SSDs such as GC (Garbage Collection) can modify the mapping relationship between LBA and PBA (Physical Block Address) without notifying the host. In this case, even if some data are located on contiguous LBAs, they can exist on dis-contiguous PBA pages, resulting in internal resource conflicts. As with other defragmentation tools, the current version of FragPicker is also unable to detect such cases. Specifically, FragPicker relies on `filefrag` program for fragmentation detection. However, `filefrag` is not designed to detect PBA fragmentation. In the future, we are planning to extend FragPicker to address such device-level fragmentation utilizing open-channel SSDs [22].

Here is a summary of the contributions of this paper. First, we clarified the rationale behind performance degradation incurred by fragmentation on modern storage devices considering their internal mechanisms, using an experimental and statistical approach. Second, we proposed a new defragmentation tool, FragPicker, for modern storage devices. FragPicker minimizes the amount of I/Os while achieving comparable performance improvements to the conventional schemes, by means of comprehensive analyses on I/O activities and efficient defragmentation. As a result, FragPicker reduces the execution time and performance interference of defragmentation while minimizing storage wear-out, compared with the conventional tools. In addition, FragPicker is able to target particular applications/users for defragmentation via the BCC function, which monitors I/O activities of only specific applications/users. Third, the analysis phase is lightweight, and thus we could not observe a noticeable performance decline in our experiments with Flash SSDs. The highest overhead induced by the analysis phase was less than 2% on the Optane SSD.

Finally, FragPicker is implemented in the userspace without directly utilizing any filesystem internal functions and modifying kernel codes. Instead, FragPicker adopts the commonly used techniques. For example, to monitor I/O activities on the system call layer, FragPicker utilizes Linux Extended BPF (eBPF) Tracing Tools [17]. Additionally, the `filefrag` program [10] and `fallocate` [9] are used for fragmentation checking and block allocation, respectively. This filesystem-agnostic attribute of FragPicker can help new filesystems-in-progress evaluate the fragmentation and defragmentation effect, especially regarding aging tests. Additionally, conventional filesystems that lack user-friendly defragmenters can readily utilize FragPicker without expensive engineering costs. The necessary features to use FragPicker are i) eBPF support, ii) filesystem-level consistency, and iii) support for `fallocate` and `filefrag`. Additionally, since FragPicker relies on user-level I/O libraries for migration, the filesystem should allocate a contiguous LBA region for a single write operation if possible.

Here is a summary of the limitations of our work. First, FragPicker monitors I/O activities on the system call layer. Therefore, the page cache effect on I/Os might not be fully considered. The page cache handles I/Os under the system call layer. Thus, some I/Os might not reach the storage and instead be serviced by the page cache. In this case, the critical data to the performance can be a bit different from our analysis. However, we still believe that monitoring I/Os on the system call layer is the right choice. The characteristics of I/Os from the applications change as they experience various kernel layers, such as journaling. Additionally, since the size of the page cache varies depending on the available memory size of the system, monitoring I/Os under the page cache can lose idempotency. Therefore, the system call layer is the adequate layer to precisely monitor the I/O activities of applications.

Second, when the system has insufficient free space or the free space is fragmented, FragPicker might fail to defragment. However, this problem occurs in other existing defragmenters as well [8]. Rather, we believe that FragPicker can indirectly reduce free space fragmentation because it migrates a smaller amount of data than other tools. Finally, since FragPicker does not consider the *frag distance*, FragPicker can increase the tail latency on devices that have seek time, such as HDDs. Therefore, we do not recommend using FragPicker on such devices.

7 Conclusion

In this paper, we experimentally analyzed the performance degradation caused by fragmentation on modern storage devices using a statistical approach. As a result, we discovered that its main cause is *request splitting*. With this observation, we proposed a new defragmentation tool, FragPicker. FragPicker analyzes the I/O characteristics of applications and identifies the pieces of data that are critical to the performance. Additionally, FragPicker performs efficient defragmentation using the analysis results, by migrating only necessary blocks of a file for the sake of preventing *request splitting*. Experimental results show that FragPicker significantly improves I/O performance while minimizing the amount of I/Os for defragmentation. FragPicker can be easily deployed to most systems, because it is not designed for a particular filesystem and does not include any modification of kernel codes.

Acknowledgments

We thank our shepherd, Ana Klimovic, and the anonymous reviewers for their invaluable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays)

References

[1] 2021. blktrace. https://linux.die.net/man/8/blktrace.

[2] 2021. btrfs-filesystem. https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs-filesystem.

[3] 2021. Consumer SSD Performance. https://www.samsung.com/semiconductor/minisite/ssd/support/faqs-03/.

[4] 2021. DebugFS. https://www.kernel.org/doc/html/latest/filesystems/debugfs.html.

[5] 2021. defrag.f2fs. https://manpages.debian.org/testing/f2fs-tools/defrag.f2fs.8.en.html.

[6] 2021. Defraggler. https://www.ccleaner.com/defraggler.

[7] 2021. disk-defrag. https://www.auslogics.com/en/software/disk-defrag/.

[8] 2021. e4defrag. http://manpages.ubuntu.com/manpages/bionic/man8/e4defrag.8.html.

[9] 2021. fallocate. https://man7.org/linux/man-pages/man2/fallocate.2.html.

[10] 2021. filefrag. https://linux.die.net/man/8/filefrag.

[11] 2021. Flexible I/O Tester. https://github.com/axboe/fio.

[12] 2021. Frequently Asked Questions for Intel® Solid State Drives. https://www.intel.com/content/www/us/en/support/articles/000006110/memory-and-storage.html.

[13] 2021. How to defrag hard drive. https://www.crucial.com/articles/pc-users/how-to-defrag-hard-drive.

[14] 2021. How to defrag your drives the right way. https://www.auslogics.com/en/articles/how-to-defrag/.

[15] 2021. Intel® Optane™ SSD 905P Series for Demanding Storage Workloads. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/enthusiast-ssds/optane-ssd-905p-brief.html.

[16] 2021. iotop. https://linux.die.net/man/1/iotop.

[17] 2021. Linux Extended BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html.

[18] 2021. The Linux Journalling API. https://www.kernel.org/doc/html/v5.7/filesystems/journalling.html.

[19] 2021. Ways to improve your computer's performance. https://support.microsoft.com/en-us/windows/ways-to-improve-your-computer-s-performance-c6018c78-0edd-a71a-7040-02267d68ea90.

[20] 2021. What Condusiv's Diskeeper Does for Me. https://condusiv.com/what-condusivs-diskeeper-windows-does-for-me.

[21] A Garcia Asuero, Ana Sayago, and AG Gonzalez. 2007. The correlation coefficient: An overview. *Critical reviews in analytical chemistry* 36, 1 (Jan. 2007), 41–59.

[22] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. USENIX FAST*. 359–374.

[23] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. ACM SIGMETRICS*. 181—-192.

[24] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. 2017. File systems fated for senescence? nonsense, says science!. In *Proc. USENIX FAST*. 45–58.

[25] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. 2019. Filesystem aging: It's more usage than fullness. In *Proc. USENIX HotStorage*. 1–7.

[26] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert. 2006. The effects of filesystem fragmentation. In *Proc. OLS*. 193–208.

[27] Congming Gao, Liang Shi, Kai Liu, Chun Jason Xue, Jun Yang, and Youtao Zhang. 2020. Boosting the performance of ssds via fully exploiting the plane level parallelism. *IEEE Trans. Parallel Distrib. Syst.* 31, 9 (2020), 2185–2200.

[28] Jiayang Guo, Yiming Hu, Bo Mao, and Suzhen Wu. 2017. Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance. In *Proc. IEEE IPDPS*. 1184–1193.

[29] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[30] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proc. USENIX ATC*. 759–771.

[31] Bryan Harris and Nihat Altiparmak. 2020. Ultra-Low Latency SSDs' Impact on Overall Energy Efficiency. In *Proc. USENIX HotStorage*. 1–8.

[32] Dan He, Fang Wang, Hong Jiang, Dan Feng, Jing Ning Liu, Wei Tong, and Zheng Zhang. 2014. Improving Hybrid FTL by Fully Exploiting Internal SSD Parallelism with Virtual Blocks. *ACM Trans. Archit. Code Optim.* 43 (Dec. 2014), 1–19.

[33] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proc. ACM EuroSys*. 127–144.

[34] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-Based Solid State Drives. In *Proc. ACM SYSTOR*. 1–9.

[35] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 2013. Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance. *IEEE Trans. on Comput.* 62, 6 (2013), 1141–1155.

[36] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. 2014. Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and Erase Scaling. In *Proc. USENIX FAST*. 61–74.

[37] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. 2018. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Trans. on Mobile Computing* 18, 9 (2018), 2062–2076.

[38] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. 2016. An empirical study of file-system fragmentation in mobile storage systems. In *Proc. USENIX HotStorage*. 1–5.

[39] Xavier Jimenez, David Novo, and Paolo Ienne. 2014. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *Proc. USENIX FAST*. 47–59.

[40] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. *SIGMETRICS Perform. Eval. Rev.* 41, 1 (June 2013), 203–216.

[41] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. 2018. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *Proc. USENIX ATC*. 691–703.

[42] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. 2019. Storage gardening: Using a virtualization layer for efficient defragmentation in the wafl file system. In *Proc. USENIX FAST*. 65–78.

[43] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. 2020. Countering Fragmentation in an Enterprise Storage System. *ACM Trans. Storage* 15, 4 (Feb. 2020), 1–35.

[44] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. 2008. Ext4 block and inode allocator improvements. In *Proc. OLS*. 263–274.

[45] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proc. USENIX FAST*. 273–286.

[46] Sehwan Lee, Bitna Lee, Kern Koh, and Hyokyung Bahn. 2011. A Lifespan-Aware Reliability Scheme for RAID-Based Flash Storage. In *Proc. ACM SAC*. 374–379.

[47] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proc. OLS*. 21–33.

[48] H. Gilbert Miller and Peter Mork. 2013. From Data to Decisions: A Value Chain for Big Data. *IT Professional* 15, 1 (2013), 57–59.

[49] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2021. *Introduction to linear regression analysis*.

[50] Jonggyu Park and Young Ik Eom. 2020. Anti-Aging LFS: Self-Defragmentation With Fragmentation-Aware Cleaning. *IEEE ACCESS* 8 (2020), 151474–151486.

[51] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. 2016. File defragmentation scheme for a log-Structured file system. In *Proc. ACM SIGOPS APSys*. 1–7.

[52] Daniel A. Reed and Jack Dongarra. 2015. Exascale Computing and Big Data. *Commun. ACM* 58, 7 (June 2015), 56––68.

[53] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Trans. on Storage* 9, 3 (2013), 1–32.

[54] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *Proc. USENIX FAST*. 67–80.

[55] Keith A. Smith and Margo I. Seltzer. 1997. File system aging—increasing the relevance of file system benchmarks. In *Proc. ACM SIGMETRICS*. 203––213.

[56] Keith A. Smith and Margo I. Seltzer. 1997. File System Aging—Increasing the Relevance of File System Benchmarks. *SIGMETRICS Perform. Eval. Rev.* 25, 1 (June 1997), 203–213.

[57] JEDEC Standard. 2012. Embedded multimedia card (eMMC), electrical standard 4.51. *JEDEC Solid State Technology Association* (2012).

[58] Richard Taylor. 1990. Interpretation of the Correlation Coefficient: A Basic Review. *J. Diag. Med. Sonogr.* 6, 1 (1990), 35–39.

[59] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of intel optane SSD. In *Proc. USENIX HotStorage*. 1–8.

[60] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2020. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 5, 1 (Feb. 2020), 1–28.

[61] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. 2013. HEC: Improving Endurance of High Performance Flash-Based Cache Devices. In *Proc. ACM SYSTOR*. 1–11.

[62] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, and Shiyun Tu. 2020. ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees. In *Proc. DATE*. 1061–1066.