# FORESEE: An Effective and Efficient Framework for Estimating the Execution Times of IO Traces on the SSD

Yoonsuk Kang, Yong-Yeon Jo, Jaehyuk Cha, Wan D. Bae, Wonjun Lee, and Sang-Wook Kim, *Member, IEEE*

**Abstract**—If we had the performance information of every application on every SSD, it would be very beneficial to both SSD users and SSD manufacturers. For SSD users, they can buy the SSD that is fastest for the most frequently using applications; for SSD manufacturers, they can figure out the strength and weakness of their SSD for every application. Toward this end, this article proposes a framework named FORESEE that estimates accurately the execution time of a given IO trace (i.e., *query IO trace*) of a given application on a target SSD *without its actual execution*. FORESEE is developed based on the observation that *if two IO traces are similar to each other in their IO behavior, their execution times tend to be similar when they are executed on the same SSD*. In FORESEE, the execution time of a query IO trace is estimated by using the execution times of the IO traces in a database similar to the query IO trace. Our technical contributions in FORESEE are as follows: (1) we propose a goodness function that efficiently evaluates the quality of sets of features that are used to measure the similarity of IO traces; (2) we propose a DB structure and a searching method for efficiently searching for similar IO traces to a query IO trace; (3) we propose an aggregation method that aggregates the execution times of similar IO traces to a query IO trace for accurately estimating the execution time of the query IO trace; and (4) we verify the effectiveness of FORESEE via extensive experiments by using real-world application IO traces. According to the results, the Pearson correlation coefficient (PCC) of the actual execution time and the estimated execution time by FORESEE is found to be 0.87, indicating FORESEE estimates the execution time accurately.

**Index Terms**—Execution time estimation, IO traces, solid-state drives

## 1 Introduction

A solid-state drive (SSD) is a relatively new type of storage device that uses NAND flash memory chips for storing data. Compared to a hard-disk drive (HDD), an SSD provides higher bandwidth, lower latency, and longer life time [1], [2]. As a result, SSDs have been rapidly replacing HDDs, as a next-generation storage device. In the early days, the usage of SSDs was limited due to their high price; however, thanks to recent advances in NAND flash memory technology, manufactures have lowered development costs of SSDs, which subsequently has reduced the prices of SSDs [3], giving users more opportunities to use them. As demand for SSD increases, various kinds of SSDs are being produced by various manufacturers.

On users' side, a user wants the SSD that provides the best performance on the applications frequently executed in his/her environment; they want to know which SSD performs best for these applications. On manufacturers' side, they aim to develop SSDs that perform better than their competitors on as many applications as possible, and they want to figure out which applications perform poorly on their SSD, in order to improve its overall performance by fixing the problem.

Assuming that $n$ SSDs and $m$ applications are given, a user wants to know *(Q1) the performance of his/her own application on each of n SSDs (n:1)*, while a manufacturer wants to know *(Q2) the performances of m applications on its SSD (1:m)*. *If we know the performances of all the applications on all the SSDs (n:m)*, we can provide the answers for *(Q1)* and *(Q2)* to both of users and manufacturers.

SSD manufacturers release simple performance specifications of their SSDs in terms of bandwidth, latency, and input/output operations per second (IOPS). The performance comparison between different manufacturer' SSDs with this information may not be that meaningful because even though the performances of SSDs obtained from this information are similar, the performances of a specific application on the SSDs may be different. In order to compare the performances of the SSDs, the performance of the SSD is measured by using *benchmarks* in general [4], [5]. Although a benchmark is designed to show the performance of executing various types of IO patterns on the SSD [5], the performance of the SSD measured by a benchmark may not always cover the performance of all of the applications on the SSD.

A simple approach to know the performance of every application on every SSD is to *run* every application on every SSD and *measure* its execution time. However, this approach

- *Yoonsuk Kang, Yong-Yeon Jo, Jaehyuk Cha, and Sang-Wook Kim are with the Department of Computer Science, Hanyang University, Seoul 04763, South Korea. E-mail: {dyskang, jyy0430, chajh, wook}@hanyang.ac.kr.*
- *Wan D. Bae is with the Department of Computer Science, Seattle University, Seattle, WA 98122 USA. E-mail: baew@seattleu.edu.*
- *Wonjun Lee is with the School of Cybersecurity, Korea University, Seoul 02841, South Korea. E-mail: wlee@korea.ac.kr.*

*Manuscript received 27 Mar. 2019; revised 15 Sept. 2020; accepted 1 Nov. 2020. Date of publication 16 Nov. 2020; date of current version 8 Nov. 2021. (Corresponding author: Sang-Wook Kim.) Recommended for acceptance by F. Cappello. Digital Object Identifier no. 10.1109/TC.2020.3038189*

0018-9340 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

is infeasible in practice because it requires a lot of cost and time. In this paper, we aim to estimate the performance of a target application on an SSD *without its actual execution*. This estimation, if accurate, provides users and manufacturers with the answers for *(Q1)* and *(Q2)*, respectively.

Unfortunately, manufacturers do not open detailed mechanisms of their SSDs [6]. Rather, as already mentioned, they release only simple performance specifications. With only this information, it is difficult to describe how those SSDs are different from each other in terms of mechanisms and what conditions make their behaviors predictable. What we do in this paper is to estimate the execution time of a query IO trace on an *arbitrary SSD whose technical mechanism is not known* (i.e., regarding the SSD as a black box) by using the execution times of IO traces that have been performed on the SSD in the past, rather than estimating the execution time of a query IO trace on a *specific SSD whose technical mechanism is known*. This approach is beneficial in practice because the execution time of a query IO trace on an SSD can be estimated without requiring (1) the technical mechanism of the SSD nor (2) knowing the performance information of the SSD.

An important observation in the literature has shown that *if two IO traces of applications are similar to each other in terms of an IO pattern, their execution times tend to be similar when they are executed on the same SSD* [7]. Based on this observation, we estimate the execution time of a query IO trace $q$ as follows: assuming that an IO trace $s$ has an IO pattern very similar to that of $q$ and has the execution time of $t_s$, the execution time of $q$, $t_q$, can be estimated as $t_s$.

In this paper, we propose FORESEE[1] (**F**ramework f**OR** **E**stimating **S**SD p**E**rformance **E**fficiently) as a fast and accurate solution to the problem of estimating the execution time of a query IO trace on a target SSD without its actual execution. We introduce a new concept called an *IO window*, a sub-IO trace of a *fixed length* to solve the problems caused by IO traces of different lengths. FORESEE first extracts query IO windows from a query IO trace and estimates the execution time of each IO window by using the known execution times of the DB IO windows stored in a database (DB). Finally, it estimates the execution time of the entire query IO trace by aggregating the estimated execution times of its query IO windows. This way of estimation enables us to estimate the execution time of a query IO trace of an arbitrary length.

The proposed framework consists of the following three components: (1) *feature set evaluation*, (2) *DB construction*, and (3) *execution time estimation*. Feature set evaluation is the component that evaluates sets of features for selecting the final set of features used to measure the similarity of IO windows; DB construction is the component that stores the feature values and the execution time of each IO window in the DB; execution time estimation is the component that first estimates the execution times of the query IO windows by using the execution times of their similar DB IO windows and finally estimates the execution time of the query IO trace by aggregating the estimated execution times of those query IO windows.

1. The initial idea of this paper was introduced with some preliminary results of evaluation at ACM CIKM 2017 as a short (four pages) paper (i.e., conference version) [8]. This paper (i.e., journal version) is its extended version written for the archival purpose in a journal.

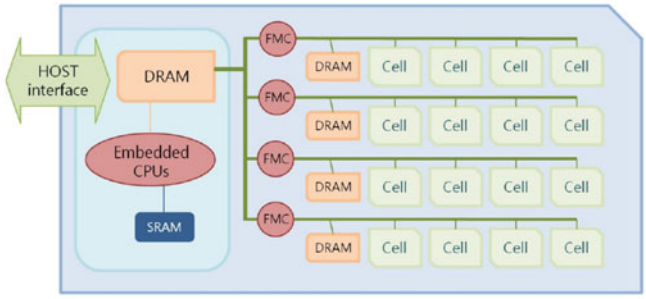



Fig. 1. Overall architecture of a SSD.

For validating the effectiveness of FORESEE, we estimate the execution times of query IO traces and measure their accuracy. As an accuracy measure, we use the Pearson correlation coefficient (PCC) to measure the correlation between two sets of the actual and estimated execution times [9]. The PCC of FORESEE is shown 0.87, which is much higher than 0.17, that of the baseline method.

Our key contributions are summarized as follows:

- We propose a framework for solving the problem of estimating the execution time of a query IO trace on the SSD without its actual execution.
- We address how to determine a set of features that are used to measure the similarity of IO traces, and propose an efficient method to find such a set of features.
- We propose a database structure and a searching method to find similar IO traces based on the determined set of features.
- We propose an aggregation method to estimate accurately the execution time of a query IO trace.
- We verify the effectiveness of FORESEE, the proposed framework, through extensive experiments.

The rest of this paper is organized as follows: Section 2 presents the background and motivation of our work. Section 3 presents the overview of the proposed framework. Section 4 describes in detail the components of the proposed framework. Section 5 presents and analyzes the evaluation results on the proposed framework. Section 6 summarizes and concludes this paper.

## 2 Background and Motivation

Fig. 1 illustrates the general architecture of an SSD with its major components [10], [11], [12], [13]: NAND flash memory chips, flash memory controllers (FMCs), embedded CPUs, and DRAMs. A *NAND flash chip* stores data permanently inside and a *flash memory controller* manages multiple NAND flash memory chips. A *DRAM* stores data temporarily to read/write from/into NAND flash chips. 8-16 NAND flash memory chips, a flash memory controller, and DRAM are organized into a *channel*. Typically, an SSD contains 8-32 channels.

An embedded CPU manages the channels by using firmware called flash translation layer (FTL) [10]. FTL performs two functions: *address mapping* and *wear leveling*. *Address mapping* is a function that maps a logical address to a

physical address when processing IO requests, thereby processing IO updates efficiently [14], [15], [16], [17], [18]. *Wear leveling* is a function that makes writing operations distributed over entire blocks in the SSD, thereby ensuring long lifetime of the SSD [15].

Since SSDs read/write data by the electronic principle while the HDDs read/write data by the magnetic principle, they provide faster bandwidth, lower latency, and longer lifetime than HDDs [19], [20], [21]. As a result, SSDs are becoming popular storage devices, replacing HDDs. Manufacturers provide various kinds of SSDs on the market.

As mentioned in the previous section, both SSD users and manufacturers want to know the performances of the SSDs for their own purpose. A simple approach to know the performance of every application on every SSD is to run and measure all applications on all SSDs. This approach, however, requires (1) purchasing all the applications and all the SSDs, (2) installing all the applications on all the SSDs, (3) running all the applications on all the SSDs, and (4) measuring their performances, which is infeasible in practice.

Sometimes, obtaining an *approximated* answer with reasonably high accuracy in short time or with low cost is preferred, rather than obtaining an *exact* answer in much long time or with much high cost [22]. For example, in the area of database, when a query is issued to a database, it can be processed in a large number of different execution plans. A query optimizer determines the execution plan whose query processing time is the shortest among those plans. For measuring the processing time of each plan, it estimates the approximated processing time, rather than measuring the exact processing time. It finally processes the query with the plan whose estimated processing time is the shortest [22].

Our goal is to estimate the execution time of each application accurately on a given SSD *without its actual execution*. If the execution time is accurately estimated, the consumers and manufacturers of the SSD can easily get the performance information of the application run on the SSD.

## 3 Framework Overview

In this section, we present the overview of FORESEE. The key idea is to estimate the execution time of a query IO trace by using the execution times of other IO traces in a DB having similar IO patterns to the query IO trace. The sizes (i.e., lengths) of entire IO traces vary from one application to another. This causes a problem that there are few IO traces that are really similar to the query IO trace. To solve this problem, we define a new concept called an *IO window* whose length is fixed. FORESEE extracts a series of query IO windows from a query IO trace. Next, it estimates the execution time of each query IO window and finally estimates the execution time of the entire query IO trace based on the estimated execution times of query IO windows.

The execution time of an IO trace extracted from an application is determined by the characteristics (i.e., IO pattern) of the IO trace; thus, if the IO pattern of an IO trace changes, the execution time of the IO trace would change. In FORESEE, however, the DB IO traces used to estimate the execution time of a query IO trace do not necessarily need to be those IO traces extracted from the application that the query IO trace belongs to. Rather, the DB IO traces extracted from any applications can be used in this case, if their IO patterns are similar to that of the query IO trace. Therefore, even when the IO pattern of the query IO trace changes over the time, FORESEE can estimate the execution time of the query IO trace if there exist such DB IO traces having similar IO pattern to that of the query IO trace.

In addition, we note that FORESEE first estimates the execution times of query IO windows and then estimates the execution time of the entire query IO trace based on those estimated execution times of query IO windows, instead of estimating the execution time of the entire query IO trace directly. The possibility of having the IO windows similar to the query IO window in terms of the IO pattern will be much larger than that of having IO traces similar to the query IO trace. Therefore, since FORESEE DB is more likely to have the DB IO windows similar to the query IO window, it could estimate accurately the execution times of both query IO windows and query IO traces in this way.

Fig. 2 shows the overall architecture of FORESEE composed of three components: *feature set evaluation*, *DB construction*, and *execution time estimation*. Feature set evaluation is the component that evaluates sets of features for selecting $k$ features for the final set of features for an IO window (i.e., final feature selection_red box with dotted lines), used for computing the similarity between IO windows. The input and output of this component are DB IO traces in the raw data and the final set of features, respectively. This component is composed of *window feature extraction* and *evaluation* steps. The process of each step will be discussed in detail in Section 4. The final set of features thus obtained is used in DB construction and execution time estimation (arrows with dotted lines).

DB construction is the component that builds a FORESEE DB storing features of IO traces and their execution times (i.e., DB construction_dotted line arrow). The input and output of this component are DB IO traces in the raw data and FORESEE DB which stores a number of pairs of a feature vector $w_i = (f_i^1, f_i^2, \ldots, f_i^k)$ of $i$th IO window and its execution time $t_{w_i}$, respectively. DB construction is composed of *window feature extraction* and *storage* steps. FORESEE DB thus constructed is used in execution time estimation. Feature selection and DB construction are preprocessing processes that should be performed *before* execution time estimation.

Execution time estimation is the component that estimates the execution time of a query IO trace (i.e., execution time estimation_black box). The input and output of this component are a query IO trace and its estimated execution time, respectively. Execution time estimation is composed of *window feature extraction*, *window matching*, and *aggregation* steps.

In FORESEE DB, a large number of pairs of an IO window with its feature vector and its execution time are stored for each individual SSD model. Also, this FORESEE DB will be continuously updated with more pairs added by SSD manufacturers and users even after its initial release. In order to provide more accurate estimation for each SSD model, its manufacturer will update the DB whenever more execution results are obtained. Also, an SSD user can add the execution results obtained with the workload on her/his own SSD model to share this information with others. We note, however, that this paper aims to address technical

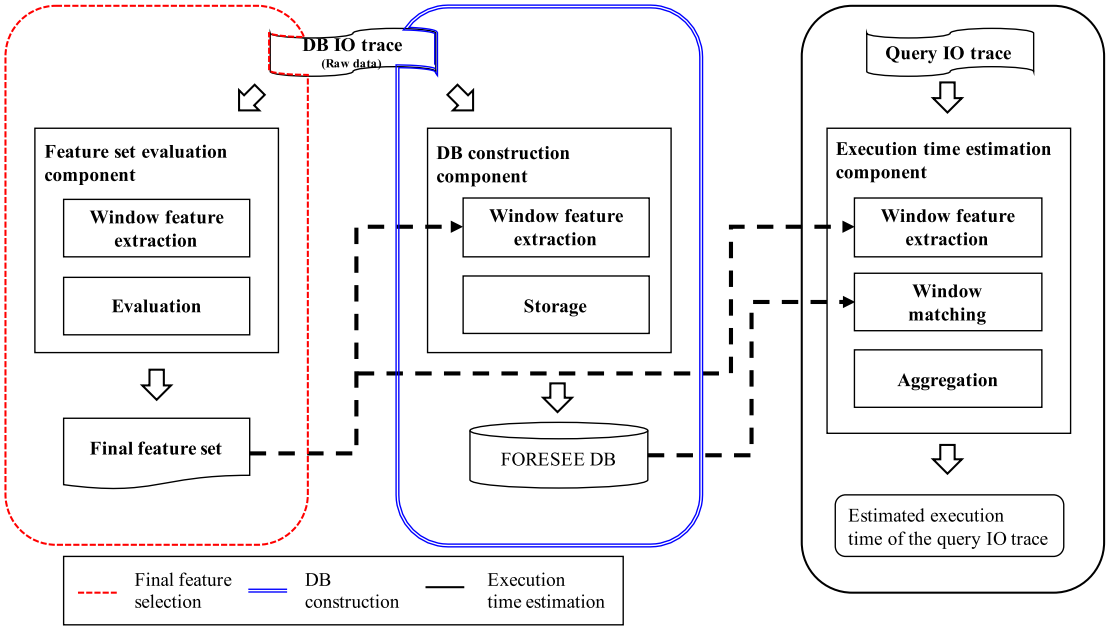



Fig. 2. Overview of FORESEE.

issues related to FORESEE, thus leaving the broader spectrum of possible service scenarios for FORESEE open.

When FORESEE is released first, its DB is unlikely to have sufficient data inside, incurring relatively low accuracy in estimating execution times. However, as time goes by, the accuracy will be expected to increase, thanks to the continuous updates added by SSD manufacturers and users.

## 4 FORESEE: Proposed Framework

In this section, we present the three components of FORESEE (i.e., *feature set evaluation*, *DB construction*, and *execution time estimation*) in detail.

### 4.1 Feature Set Evaluation

Feature set evaluation is the component of FORESEE that evaluates a number of features sets to select the final features set, which is used to find similar IO traces via following two steps: (1) *window feature extraction* and (2) *evaluation*.

#### 4.1.1 Window Feature Extraction

If the lengths of entire IO traces are different, their execution times are usually different [7]. Therefore, in order to estimate the execution time of a query IO trace, we need the IO traces whose lengths are similar to that of the query IO trace. The lengths of IO traces in real applications, however, are (1) very long and (2) quite different to one another [23], [24]. In this case, the number of IO traces with similar lengths to that of the query IO trace should be very small. Even when the IO trace is similar to the query IO trace in length, the possibility of having the IO traces similar to the query IO trace in IO patterns will be very small. In summary, in the *matching step* in execution time estimation that searches for the IO traces similar to the query IO trace, there is a problem that it is rare to find successfully those DB IO traces that are truly similar to the entire query IO trace. This problem arises because the matching is performed in the unit of the entire query IO trace whose length is very long. In order to solve this problem, we set the unit of matching as a "*partial IO trace of a fixed length*", rather than an "*entire IO trace*". In this paper, we define *IO windows* as such partial IO traces of a fixed length.

The proposed framework constructs a FORESEE DB by extracting a series of DB IO windows from a large number of DB IO traces. Then, when a query IO trace is issued, it extracts a series of IO windows (i.e., *query IO windows*) from the query IO trace and estimates the execution time of each query IO window by using the DB IO windows that are similar to the query IO window. Finally, FORESEE estimates the execution time of the entire query IO trace by aggregating the estimated execution times of those query IO windows.

In order to extract IO windows from an IO trace, two extraction methods can be used: (1) *sliding window extraction* and (2) *disjoint window extraction*. The sliding window extraction method extracts an IO window of length $w$ starting from every IO request in the IO trace. This makes two adjacent IO windows overlapped, which means an IO request can be included in multiple adjacent IO windows. The disjoint window extraction method extracts an IO window of length $w$ starting from an IO request that appears right after the last IO request of its previous IO window. This method makes all the windows disjoint to one another, thereby having an IO request belong to only one IO window.

Fig. 3 shows the results of extracting IO windows of 3 KB length from IO trace $A$ whose length is 6 KB by (a) *sliding window extraction* and (b) *disjoint window extraction*. In sliding window extraction, the first IO window consists of the first and second IO requests of $A$; the second IO window consists of the second and third IO requests of $A$; and the last IO window consists of the third and fourth IO requests of $A$. In disjoint extraction, the first IO window consists of the first and second IO requests of $A$; the second IO window consists of the third and fourth IO requests of $A$; the two IO windows, here, do not have any IO request in common.

If the number of IO windows stored in FORESEE DB increases, it is more likely that we find DB IO windows that

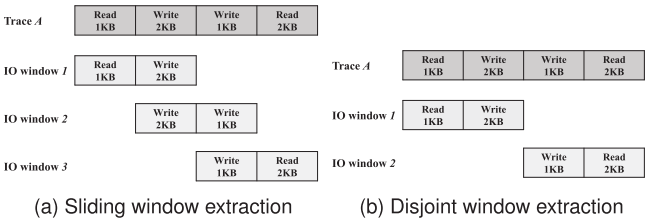



Fig. 3. Extracting IO windows by two methods.

are similar to the query IO window. We note the total number of IO windows extracted by sliding window extraction is larger than that of IO windows extracted by disjoint window extraction. Therefore, in this paper, we decide to employ sliding window extraction to extract DB IO windows from DB IO traces in order to have a more number of DB IO windows in FORESEE DB.

When performing matching between a query IO trace and a DB IO trace, two different ways of window extraction are employed by following [25], [26]. Since, sliding window extraction is used in constructing FORESEE DB, disjoint window extraction is employed to extract query IO windows from the query IO trace.

#### 4.1.2 Evaluation

FORESEE estimates the execution time of the query IO window by querying the top-$p$ DB windows that are most similar to the query IO window in terms of the IO pattern. When $k$ features are given, as shown in Equation (1), we measure the similarity of two feature vectors of two IO windows $w_i = (f_i^1, f_i^2, \ldots, f_i^k)$ and $w_j = (f_j^1, f_j^2, \ldots, f_j^k)$ by using the *inverse* of the euclidean distance to judge how much $w_i$ and $w_j$ are similar in terms of the IO pattern

$$sim(w_i, w_j) = \frac{1}{\sqrt{\sum_{m=1}^{k} (f_i^m - f_j^m)^2}}. \quad (1)$$

The reasoning behind this is based on the observation that *the execution times of two IO traces (i.e., two IO windows here) are similar when their corresponding feature values are similar* [7].

To find a feature set that satisfies the above observation, we first find a set of candidate features that may affect the execution time on the SSD. Among those candidate features, *some* of them could satisfy the observation. Therefore, among all possible combinations of those candidate features, we identify the best combination of candidate features as the final feature set for FORESEE.

Table 1 shows 20 candidate features of IO windows considered in this paper. The number of IO requests, the size of IO requests (average and standard deviation), the interval time of two adjacent IO requests (average and standard deviation), the range of the logical addresses (average and standard deviation), the ratio of the number of read/write IO requests to that of all IO requests, the ratio of the number of sequential/random read IO requests to that of all read IO requests, and the ratio of the number of sequential/random write IO requests to that of all write IO requests are those features well-known to affect the performance of the SSD [3], [27].

The ratio of the *number* of read IO requests to that of write IO requests is known to affect the performance of the SSD in the literature. However, if the ratios of the *amount* of read IOs to that of write IOs in two IO traces are different, their execution times may be different even if the ratios of the number of read IO requests to that of write IO requests in the two IO traces are identical. For instance, there are two IO traces $A$ and $B$, both of which have 5 read IO requests and 5 write IO requests. The amounts of read and write IO requests in $A$ are 9 and 1 MB, respectively, while the amounts of read and write IO requests in $B$ are 1 and 9 MB, respectively. In this case, the execution times of $A$ and $B$ would be considerably different. In order to consider the ratio of the amounts of read and write IO requests, we add the features from 14 through 19 in Table 1.

HDDs perform *in-place updates* that write 'will-be-updated data' to the space of the SSD where the original data were written, while the SSD performs an *out-place update* that writes 'will-be-updated data' to another free space of the SSD [13]. Thus, an update involves a large number of write and erase operations; therefore, if the number of updates in an IO window increases, it is more likely that the execution time of the IO window on the SSD becomes larger, indicating the updates significantly affect the performance of the SSD. Therefore, we add the ratio of updates to writes as a new candidate feature in Table 1.

There are a large number of feature sets composed of all possible combinations of the candidate features in Table 1. Of course, we can use all of the 20 candidate features for the final feature set. We know that FORESEE estimates the execution time of the query IO window by using the execution

TABLE 1
Candidate Features

| Feature ID | Name | Meaning | Feature ID | Name | Meaning |
|---|---|---|---|---|---|
| 1 | IO count | The # of IOs | 11 | Sequential write ratio (ioc) | Sequential write ratio in terms of # of IOs |
| 2 | IO size (avg) | Average of IO sizes | 12 | Random read ratio (ioc) | Random read ratio in terms of # of IOs |
| 3 | IO size (std) | Standard deviation of IO sizes | 13 | Random write ratio (ioc) | Random write ratio in terms of # of IOs |
| 4 | IO interval (avg) | Average of IO interval times | 14 | Read ratio (iob) | Read ratio in terms of IO amount |
| 5 | IO interval (std) | Standard deviation of IO interval times | 15 | Write ratio (iob) | Write ratio in terms of IO amount |
| 6 | IO locality (avg) | Average of IO logical block addresses | 16 | Sequential read ratio (iob) | Sequential read ratio in terms of IO amount |
| 7 | IO locality (std) | Standard deviation of IO logical block addresses | 17 | Sequential write ratio (iob) | Sequential write ratio in terms of IO amount |
| 8 | Read ratio (ioc) | Read ratio in terms of # of IOs | 18 | Random read ratio (iob) | Random read ratio in terms of IO amount |
| 9 | Write ratio (ioc) | Write ratio in terms of # of IOs | 19 | Random write ratio (iob) | Random write ratio in terms of IO amount |
| 10 | Sequential read ratio (ioc) | Sequential read ratio in terms of # of IOs | 20 | Overwrite ratio | Update ratio |

times of DB IO windows similar to the query IO window in IO pattern. The similarity of execution times of similar IO windows is considerably affected by the candidate features employed in finding the similar IO windows. Thus, FORESEE need to use such a feature set that makes the similar IO windows chosen by itself have execution times as much similar as possible. In order to determine the final feature set best for FORESEE, the evaluation step evaluates every candidate feature set by using a *goodness function*.

The goodness function shown in Equation (2) below is intended to evaluate quantitatively a given feature set $F_i$ based on the execution times of $p$ similar IO windows identified by the feature set. For each IO window $w_j$ and it's top-$p$ most similar IO windows ($w_{j_s} \mid w_{j_s} \in W$) extracted from DB IO traces in the raw data according to $F_i$, the goodness function first measures the average of the differences between the execution times ($t_{w_j}$-$t_{w_{j_s}}$) of $w_j$ and every $w_{j_s}$. It then repeats the same process for all of the IO windows in the raw data. Finally, the goodness function averages the differences over all IO windows. The inverse of this value becomes the final score of the goodness function (GF-score) for the feature set

$$GF(F_i) = \frac{1}{\frac{1}{|W|}\sum_{w_j \in W}\frac{1}{|p|}\sum_{k \in p}\left|t_{w_j} - t_{w_k}\right|}. \quad (2)$$

A higher GF-score indicates similar IO windows have more similar execution times, which means that the execution times of IO windows are similar if they are similar in terms of the given feature set.

When $n$ IO windows are extracted from DB IO traces in the raw data, the complexity of finding the top-$p$ most similar DB IO windows to a given IO window is O($n$), assuming GF uses a sequential scan; thus, the complexity of repeating it for all the IO windows in FORESEE DB is O($n^2$). Similarly, the complexity of calculating execution time differences between a given IO window and its top-$p$ most similar IO windows is O($p$); thus, the complexity of repeating it for all the IO windows in the raw data is O($pn$). Therefore, the overall complexity of computing a GF-score is O($n^2$+$pn$), which is infeasible when considering that the number of DB IO windows is 3.3million in our case.

In this paper, we thus propose a new goodness function (called *goodness function*$^*$) that pursues the goal of the original goodness function with low complexity. This new function is based on the observation that, if the IO windows are grouped together according to the feature set whose GF$^*$-score is higher than that of any other feature sets, the execution times of IO windows in the same group should be more similar than that of similar IO windows grouped according to the any other feature sets. The goodness function$^*$ first makes similar IO windows grouped in terms of a feature set, and then evaluates the feature set by computing how much similar the execution times of IO windows in each group are.

To make similar DB IO windows grouped, we utilize the *k-means* algorithm, which is one of the most widely used clustering algorithms in data mining [9]. The *k*-means is the clustering algorithm most widely used in data mining, thanks to its efficiency and simplicity. This algorithm makes each data object belong to the cluster that has a centroid closest to the object [28], which makes the distances among the objects in the same cluster close to each other. This characteristic fits in our situation where we need to find IO windows that are similar to each other. For this reason, we decided to use the *k*-means for clustering of IO windows in our framework. In [27], [29], the authors show that the *k*-means algorithm provides the best clustering results among other clustering algorithms in grouping IO traces, which supports our decision as well. The *k*-means algorithm divides a set of feature vectors of all DB IO windows into $k$ groups as shown in Algorithm 1.

**Algorithm 1.** *k*-Means Algorithm

**Input:** DB IO windows, $k$ (# of clusters), $i$ (iterations)
**Output:** a set of $k$ clusters
1: Initialize $k$ random center vectors.
2: **repeat**
3: **for** each DB IO window **do**
4: Calculate the distances between the vector of the corresponding IO window and the center vectors of $k$ clusters.
5: Find the closest cluster to the DB IO window, and assign this DB IO window to the cluster.
6: **end for**
7: Recalculate the center vector of each cluster by averaging the vectors of DB IO windows included in the cluster.
8: **until** $i$ times

We measure the goodness of a given feature set for every cluster thus formed and determine the total goodness of the feature set by aggregating the goodness of the feature set for all clusters. At this time, we use the *standard deviation* of the execution times of the IO windows for measuring the goodness of a feature set for a cluster. As the standard deviation of execution times in a cluster gets smaller, the IO windows in the cluster become more similar in terms of execution times.

The numbers of IO windows in clusters may be different; therefore, we need to consider them as weights in measuring the goodness function$^*$ score (GF$^*$-score) of a feature set. Assuming $k$ clusters ($C_k$) are given in the DB IO windows extracted from DB IO traces in the raw data, as shown in Equation (3), we use the weighted sum of the standard deviations ($\sigma_j$) for all clusters, where the weight is set as the ratio of the number of IO windows in a cluster to that of all IO windows. The inverse of this value becomes the final GF$^*$-score of the given feature set

$$GF^*(F_i) = \frac{1}{\sum_{j \in k}\frac{|C_j|}{\sum_{i \in k}|C_i|}\sigma_j}. \quad (3)$$

Given $k$ clusters and $i$ iterations, the complexity of *k*-means is O($kni$). Supposing the average number of IO windows in a cluster is $\frac{n}{k}$, the complexity of computing the standard deviation of the execution times of IO windows in the cluster is O($\frac{n}{k}$); thus, the complexity of calculating the standard deviation for all $k$ clusters is O($n$). Therefore, the overall complexity of the goodness function$^*$ is O($kni$+$n$), which is much lower than O($n^2$+$pn$), that of the original goodness function, considering $k \ll n$ and $t \ll n$. In this paper, we finally use the goodness function$^*$ instead of the goodness function in the feature set evaluation component of FORESEE.

One may consider to evaluate all possible feature sets by using our goodness function*, selecting the final feature set having the highest GF*-score. Assuming $f$ candidate features are given, the number of possible combinations of features is $2^f$, which makes the problem of evaluating all possible feature sets NP-hard.

To address this problem, in this paper, we use the *greedy approach* to find the approximate solution [30], rather than the exact one. This approach obtains $BF_i$ in each step $i$, which contains $i$ candidate features. $BF_i$ is composed of $BF_{i-1}$ with one more candidate feature added. This approach first obtains the set having only one candidate feature whose GF*-score is highest as $BF_1$ among all the sets having only one candidate feature. To obtain $BF_i$ (i≥2), it measures the GF*-score of a set of one candidate feature ($cf_j$) added to $BF_{i-1}$ (i.e., $BF_{i-1} \cup \{cf_j\}$), selects $cf_k$ that makes $BF_{i-1} \cup \{cf_k\}$ get the highest GF*-score, and finally obtains $BF_i$ that contains $BF_{i-1} \cup \{cf_k\}$.

We can obtain $f$ $BF_i$s by this approach, each of which is the best set composed of $i$ features. Among them, we choose the final feature set ($FF$) that shows the highest GF*-score for FORESEE as shown in Equation (4)

$$FF = \arg\max_{1 \le i \le f} GF^*(BF_i). \tag{4}$$

This feature selection procedure is given in Algorithm 2.

**Algorithm 2.** Feature Selection Algorithm

**Input:** $f$ candidate features ($cf_j$), DB IO windows
**Output:** final feature set ($FF$)
1: Set $BF_0$ as $\phi$
2: **for** $i \leftarrow 1$ to $f$ **do**
3: **for** each $cf_j$ that $cf_j \notin BF_{i-1}$ **do**
4: Measure the GF*-score of ($BF_{i-1} \cup \{cf_j\}$) by using Equation (3)
5: **end for**
6: Select the candidate feature $cf_k$ that makes ($BF_{i-1} \cup \{cf_k\}$) get the highest GF*-score.
7: Set $BF_i$ as ($BF_{i-1} \cup \{cf_k\}$)
8: **end for**
9: Determine $FF$ by using Equation (4) among $BF_i$'s

Suppose the number of candidate features is $f$, the number of feature sets evaluated by the greedy approach is ($\frac{f(f+1)}{2}$), which is much lower than $2^f$, that of possible combinations of candidate features.[2] Also, the complexity of selecting the final feature set by using GF* and the greedy approach is O ($(kni+n) \times \frac{f(f+1)}{2}$), which is much lower than O($(pn+n^2) \times 2^f$), that of using GF for all possible feature sets. Therefore, in this paper, we use the goodness function* (i.e., rather than the goodness function) and the greedy approach (i.e., rather than the exhaustive approach) to select the final feature set for FORESEE.

## 4.2 DB Construction

DB construction is the component of FORESEE that stores IO windows (DB IO windows) in FORESEE DB via following two steps: (1) *window feature extraction* and (2) *storage*.

2. In Section 5.2.2, we also validate that the sacrifice in the quality of the result obtained by the greedy approach in insignificant.

### 4.2.1 Window Feature Extraction

As mentioned in Section 4.1.1, we extract DB IO windows from DB IO traces by using *sliding window extraction* in order to have a larger number of DB IO windows in FORESEE DB (see Section 4.1.1).

### 4.2.2 Storage

Matching is a step of execution time estimation that finds DB IO windows similar to a given query IO window. The *sequential scan* is the basic method to find the top-$p$ most similar DB IO window [31]. Since the complexity of performing the sequential scan for $n$ IO windows is O($n$). A large number of DB IO windows in FORESEE DB would make the sequential scan of the entire DB IO windows inefficient.

We reduce the complexity of matching by performing sequential scan only for some partial DB IO windows that are likely to be similar to the query IO window, excluding the rest of DB IO windows (more details will be described in the next section). If similar IO windows are grouped together and stored in FORESEE DB (say, 3.3 million in our case), this can be made easy. To this end, FORESEE uses a DB structure that has similar IO windows grouped together.

As mentioned earlier, the feature selection component performs clustering to measure the GF*-score of every feature set. The clustering result of the IO windows based on the final feature set by using the $k$-means algorithm is the DB structure that we want, because similar DB IO windows are grouped together in $k$ clusters according to the final .001 feature set.

## 4.3 Execution Time Estimation

Execution time estimation is the component of FORESEE that estimates the execution time of a query IO trace via following three steps: (1) *window feature extraction*, (2) *window matching*, and (3) *aggregation*.

### 4.3.1 Window Feature Extraction

As mentioned in Section 4.1.1, we extract query IO windows from an query IO trace by using *disjoint window extraction* (see Section 4.1.1).

### 4.3.2 Window Matching

As stated earlier, FORESEE DB consists of $k$ clusters, each of which has a number of similar IO windows. In this paper, we propose an efficient window matching method exploiting this DB structure. This method conducts sequential scan only for *partial* DB IO windows that might be similar to the query IO window instead of conducting sequential scan on *all* DB IO windows.

The proposed matching method conducts *cluster matching* first for finding a cluster that contains the IO windows that might be similar to the query IO window. In order to find such a cluster, FORESEE measures the similarity between the query IO window and *every cluster* by computing the similarity between the feature vector of the query IO window and the *center vector* of the cluster. The center vector is the average feature vector of all DB IO windows in the cluster. It then performs the sequential scan only for the IO windows

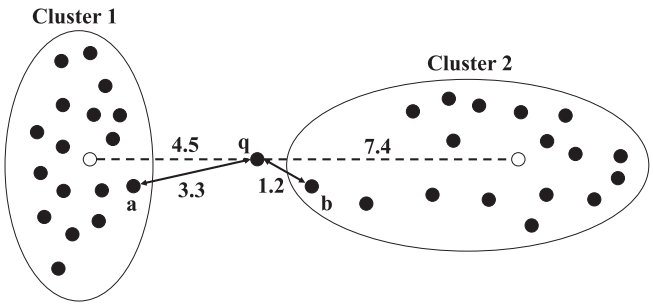



Fig. 4. An example of our proposed matching method.

in the cluster whose similarity to the query IO window is highest among all clusters.

Supposing that $n$ IO windows are stored in FORESEE DB organized with $k$ clusters and that each cluster contains $\frac{n}{k}$ IO windows, the complexity of cluster matching is O($k$) and the complexity of the sequential scan for the IO windows in a cluster is O($\frac{n}{k}$); thus, the overall complexity of the proposed window matching is O($k + \frac{n}{k}$), which is lower than O($n$), that of the sequential scan for all DB IO windows in FORESEE DB.

A problem with choosing only one cluster in cluster matching is that it may *miss* the IO windows that are actually similar to the query IO window. Fig. 4 shows this example, where FORESEE DB consists of two clusters. Each cluster stores IO windows represented by black dots in the figure. The distances of a query IO window, $q$, to center vectors of cluster 1 and cluster 2 (white dots) are 4.5 and 7.4, respectively. As mentioned in Section 4.1.2, since we use the *inverse of the euclidean distance* as the similarity measure, the low distance between the two IO windows represents high similarity between them; therefore, cluster 1, which is closer to $q$, is selected as the cluster matching result. Then, the sequential scan is performed only for the IO windows in cluster 1, determining $a$ as the IO window closest to $q$.

We note, however, the point closest to $q$ is $b$ in cluster 2 rather than $a$ since the distance between $b$ and $q$ is 1.2 while the distance between $a$ and $q$ is 3.3. However, since $b$ is not the IO window contained within cluster 1 selected in the cluster matching, it is excluded in the sequential scan. This causes FORESEE to use less similar DB IO windows in estimating the execution time of the query IO window, which subsequently leads to low estimation accuracy.

To solve this problem, our matching method searches for the top-$l$ (rather than top-one) most similar clusters and searches for top-$p$ most similar IO windows in those $l$ clusters thus found. There are some tradeoffs: if $l$ increases, the total time of the window matching increases, while the possibility of missing similar IO windows decreases; if $l$ decreases, the total time of the window matching decreases, while the possibility of missing similar IO windows increases.

#### 4.3.3 Aggregation

The aggregation step of the execution time estimation component estimates the execution time of every extracted query IO window and the execution time of the entire query IO trace via two aggregation methods: (1) an aggregation method for the query IO window and (2) an aggregation method for the query IO trace.

The top-$p$ most similar DB IO windows to a query IO window $q$ resulting from the window matching step have different similarities to $q$. In order to consider them, FORESEE estimates the execution time of the query IO window by computing the weighted average of the execution times of the top-$p$ most similar DB IO windows, where the weight is set as the similarity between a DB IO window and $q$ defined as follows:

$$t_{q_w} = \frac{\sum_{i=1}^{p} w_i t_i}{\sum_{k=1}^{p} w_k}. \tag{5}$$

Since the query IO windows are extracted from the query IO trace by using *disjoint window extraction*, FORESEE estimates the execution time of the entire query IO trace by *summing* the estimated execution times of all query IO windows as follows:

$$t_{q_t} = \sum_{i=1}^{w} t_{qw_i}. \tag{6}$$

## 5 Evaluation

In this section, we evaluate FORESEE via extensive experiments with real-world trace data. We designed our experiments for evaluation, aiming at answering the following key questions:

- Related to the goodness of feature sets:
  1) What is appropriate size of IO windows for accurate estimation?
  2) How much does the greedy approach sacrifice the accuracy in selecting the final feature set?
  3) Does GF* successfully achieve the goal that GF pursues?
  4) What are the most important features that should be included in the final feature set?
  5) If two IO windows have similar values in the final feature set, is it true that their execution times are really similar?
- Related to the effectiveness of FORESEE:
  1) How accurate is FORESEE in estimating the execution times for query IO windows?
  2) How accurate is FORESEE in estimating the execution times for entire query IO traces?

### 5.1 Experimental Environment

Our experiments were conducted on a server equipped with i7-920 2.67 GHz CPU, 12 GB main memory, and Linux 3.5.0-23. We used Hynix, Samsung, Intel, and Plextor SSDs whose performance specifications are shown in Table 2. In our experiments, TPC-C, uFLIP, and websearch IO traces are used as trace data. The TPC-C IO traces were extracted from the TPC-C benchmark that is designed to evaluate the performance of DBMSs or storage devices [32]. The uFLIP IO traces were extracted from the uFLIP benchmark that is designed to evaluate the performance of flash devices [4]. The websearch IO traces are those extracted from actual websearch engines [33]. These workloads have been used as well for evaluation in other related work [34], [35], [36], [37], [38]

TABLE 2
Performance Specifications of the Four SSDs

| | Capacity (GB) | Read Bandwidth (MB/s) | Write Bandwidth (MB/s) | 4K Rand. Read IOPS | 4K Rand. Write IOPS |
|---|---|---|---|---|---|
| **Hynix** | 120 | 510.5 | 357.9 | 97,049 | 68,710 |
| **Samsung** | 120 | 513.2 | 389.8 | 100,915 | 79,820 |
| **Intel** | 128 | 511.7 | 374.6 | 89,676 | 82,304 |
| **Plextor** | 120 | 518.9 | 325.7 | 95,411 | 67,020 |

TABLE 3
Detailed Statistics of the Dataset

| Type | Name | # of IO traces | Total amount of IO requests (GB) |
|---|---|---|---|
| **DB** | **TPC-C** | 14 | 110.6 |
| | **uFLIP** | 389 | 350.1 |
| | **Websearch** | 2 | 2.5 |
| | **Total** | 405 | 463.2 |
| **Query** | **TPC-C** | 4 | 32.4 |
| | **uFLIP** | 97 | 69 |
| | **Websearch** | 1 | 1.4 |
| | **Total** | 102 | 102.8 |
| **Total** | | 507 | 566 |

Table 3 shows the information of the entire IO traces used in our experiments. In this paper, 18 TPC-C IO traces, 486 uFLIP IO traces, and 3 websearch IO traces were used. Among them, we selected randomly 80 percent of all traces in our dataset as DB IO traces while we used the remaining 20 percent of IO traces as query IO traces. Also, we extracted DB IO windows from a DB IO trace by using sliding window extraction while we extracted query IO windows from a query IO trace by using disjoint window extraction.

## 5.2 Goodness of Features Sets

### 5.2.1 Determining the Size of IO Windows

If the size of IO windows is set too large, it is difficult to find DB IO windows similar to a query IO window, which causes large estimation errors. On the other hand, if the size of IO windows is set too small, it is easy to find DB IO windows similar to a query IO window; however, similar IO windows of this small size may have quite different execution times since the IO performance characteristics determining their execution times are unlikely to appear in such small windows. Therefore, we need to decide the size of IO windows properly for accurate estimation. Through our preliminary experiments, we examined the changes of the GF*-scores with different sizes of IO windows: we first built eight FORESEE DBs, each having IO windows of 16 to 2048 KB, and evaluated each candidate feature by using GF*.

Fig. 5 shows the GF*-scores for candidate features with different sizes of IO windows on the Hynix SSD.[3] The $x$-axis indicates candidate features, the $y$-axis indicates the sizes of IO windows, and the $z$-axis indicates the GF*-scores. The patterns of GF*-scores for the 20 candidate features were shown very similar regardless of the sizes of IO windows. Fig. 6 shows the GF*-scores with different sizes of IO windows for the 19th candidate feature, which shows the highest GF*-scores among 20 candidate features on the Hynix SSD[4]. The $x$-axis indicates the sizes of IO windows and the $y$-axis indicates the GF*-scores. When the size increases from 16 to 64 KB, the GF*-scores increase; this is because similar IO windows of a small size may have quite different execution times since the IO performance characteristics determining their execution times are unlikely to appear in such small windows. When the size increases from 64 to 512 KB, the GF*-scores are not changed. The GF*-score is highest (84.57) with the IO windows of 128 KB. When the window size increases from 512 to 2048 KB, the GF*-scores decrease: we note we select top-p DB IO windows most similar to a query IO window; as the window size gets larger, it becomes more difficult to find such DB IO windows that are sufficiently similar to the query IO window; in this situation, DB IO windows having low similarity to the query IO window will be used in estimation, causing large estimation errors. For other 19 candidate features, we observed quite similar tendencies.

In the following experiments, we set the size of IO windows fixed as 128 KB. 3,371,381 DB IO windows were extracted from the DB IO traces by sliding window extraction, while 842,847 query IO windows were extracted from query IO traces by disjoint window extraction.

### 5.2.2 Suitability of the Greedy Approach

Unlike the exhaustive approach that evaluates all possible feature sets for selecting the final feature set, the greedy approach evaluates a much reduced number of feature sets that are likely to be the final feature set. At this time, we need to validate that can we use the greedy approach in selecting the final feature set instead of the exhaustive approach.

We evaluated and compared the two performances. In the exhaustive approach, evaluating $2^{20}$ feature sets (i.e., all possible combinations of 20 features) requires a huge amount of time. We thus selected 5 features randomly from total 20 features and compared the greedy approach and the exhaustive approach on them under GF* as follows: We measured the GF*-scores of all possible feature sets by using the exhaustive approach; we sorted the feature sets in descending order of GF*-scores; we determined the final feature set by using the greedy approach and examined its rank among the sorted feature sets obtained by the exhaustive approach; we repeated this experiment five times under different sets of 5 features and averaged the 5 ranks thus

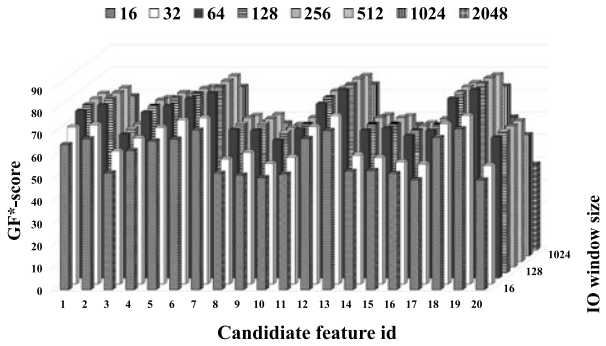



Fig. 5. GF*-scores for 20 candidate features with different IO window sizes.

3. Note that we observed similar trend results on the other three SSDs.

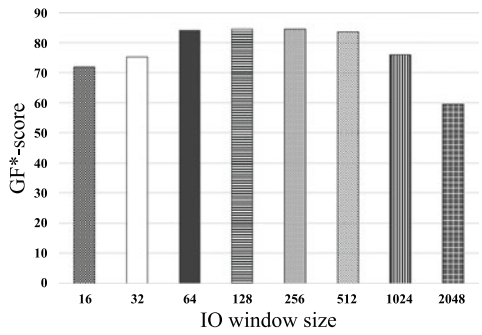



Fig. 6. GF*-scores for the 19th candidate feature with different IO window sizes.

obtained. The result shows that the average rank of the final feature set determined by the greedy approach was 3.2 out of $2^5$ possible feature sets. In summary, we can select the final feature set very efficiently by using the greedy approach without sacrificing significant amount of accuracy.

#### 5.2.3 Suitability of the GF*

In order to estimate accurately the execution time of a query IO trace, we need to select the final feature set appropriate for FORESEE. FORESEE selects the feature set whose GF*-score is higher than any other feature sets as the final one. GF* is designed to replace GF whose complexity is too high; therefore, we need to verify whether GF* actually achieves the goal of GF.

To this end, we determine the order of candidate features that are included in the final feature set by the two methods: one by using (the greedy approach + GF) and the other by using (the greedy approach + GF*). If the order of candidate features obtained by using GF and that obtained by using GF* are sufficiently similar to each other, GF* can be judged to play the same role as GF. We use the Pearson correlation coefficient (PCC) as defined in Equation (7), to determine the similarity between the two orders of candidate features. Higher correlation indicates that the order obtained by using GF* is more similar to the order obtained by using GF

$$PCC(O_{GF}, O_{GF^*}) = \frac{E\left[\left(O_{GF} - \mu_{O_{GF}}\right)\left(O_{GF^*} - \mu_{O_{GF^*}}\right)\right]}{\sigma_{O_{GF}}\sigma_{O_{GF^*}}}. \tag{7}$$

Evaluating a feature set using GF for all the DB IO windows in FORESEE DB requires too much time. We therefore evaluated a feature set after making the overall size of FORESEE DB small. We reconstructed FORESEE DB by sampling 10,000 DB IO windows randomly and measured the PCC of the two orders of candidate features. We performed this sampling 10 times to avoid bias in selection and measured the average of all the PCCs.

The GF-score can be changed if we use different numbers of similar IO windows to be used in execution time estimation; similarly, the GF*-score can be changed if we use different numbers of clusters. Thus, for GF, we varied the number of similar IO windows to 1, 2, and 3; similarly, for GF*, we varied the number of clusters to 500, 1,000, 5,000 and 10,000.

Table 4 shows the PCC of the two orders of candidate features obtained by using GF and GF* on the Hynix SSD[4]. *GF@i* indicates GF with *i* similar IO windows, and *GF*@j* indicates GF* with *j* clusters. The value of (*i*, *j*) in the table shows the PCC of the two orders obtained by using *GF@i* and *GF*@j*. The range of all the PCCs is 0.69 to 1, which indicates the two orders of candidate features obtained by GF and GF* are very similar. In addition, we observe that the PCC becomes higher as the number of similar IO windows decreases or the number of clusters increases. More detailed analysis will be described in the next section.

TABLE 4
PCC Between Two Orders Obtained by GF and GF*

| | GF* @500 | GF* @1000 | GF* @5000 | GF* @10000 |
|---|---|---|---|---|
| **GF@1** | 0.98 | 0.99 | 0.99 | 1 |
| **GF@2** | 0.69 | 0.98 | 0.98 | 0.99 |
| **GF@3** | 0.69 | 0.69 | 0.69 | 0.98 |

We also measured the execution times of the two methods in FORESEE DB of different sizes. We evaluated a set composed of 9 features on the Hynix SSD (in Table 5) in FORESEE DBs having 100, 1,000, 10,000, and 100,000 IO windows. At this time, we set the number of similar IO windows used in GF to 1 and set the number of clusters used in GF* to 10 percent of the number of IO windows in FORESEE DB.[4] Fig. 7 shows the results. The *x*-axis indicates the number of IO windows in FORESEE DB (log scale), and the *y*-axis indicates the execution time (log scale). We see that the execution time of GF* is much shorter than that of GF regardless of FORESEE DB sizes.

#### 5.2.4 Final Feature Set for FORESEE

In this section, we present the most important features to be included in the final feature set for FORESEE, which is selected by using GF* and the greedy approach. As already mentioned, the GF*-scores can be different if we use different numbers of clusters in FORESEE DB; therefore, we evaluated feature sets with different numbers of clusters in FORESEE DB (i.e., 500, 1,000, 5,000, and 10,000).

Fig. 8 shows the results of the Hynix SSD. The *x*-axis indicates the number of features in the feature set, and the *y*-axis indicates the GF*-score. The line graphs in blue, red, green, and black indicate the results of GF* with 500, 1,000, 5,000, and 10,000 clusters, respectively. In all the line graphs, the GF*-score increases sharply at the beginning, increases gradually to a certain point, and then decreases gradually after the point. The peak point of the GF*-score (120.88) appears when 9 features are used with 10,000 clusters in FORESEE DB. The features selected in the final feature set of the four SSDs are shown in Table 5. In later experiments, we used those final features sets for execution time estimation.

#### 5.2.5 Validating the Observation

We designed our FORESEE based on the observation that if the feature values of chosen feature set in two IO traces are similar, their execution times are also similar. In this section, we design and perform the following experiments to

4. We note our results in Section 5.3.1 show GF using only one similar IO window provides the best accuracy in estimating the execution time.

TABLE 5
Final Feature Sets

| | Feature IDs |
|---|---|
| **Hynix** | 2, 3, 7, 8, 10, 12, 16, 18, 19 |
| **Samsung** | 1, 2, 4, 5, 10, 12, 13, 19 |
| **Intel** | 2, 4, 5, 6, 7, 10, 11, 12, 18, 19 |
| **Plextor** | 1, 4, 5, 8, 11, 12, 15, 16, 17, 19 |

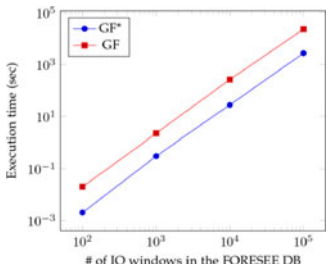



Fig. 7. Execution time of GF and GF*.

validate the observation. We first randomly select a DB IO window as a target IO window. Then, we select the top-100 most similar DB IO windows to the target IO window based on the final feature set. For all pairs of a target DB IO window and its similar DB IO window, we measure the correlation between the similarity and execution time difference. We repeat this process for 10,000 DB IO windows as target IO windows.

Figs. 9a, 9b, 9c, and 9d show the results of scatter plots for four target DB IO windows randomly selected among 10,000 DB IO windows, where each point corresponds to a pair of a target IO window and its similar DB IO window. The $x$-axis indicates the similarity between the target IO window and its similar DB IO window, and the $y$-axis indicates their execution time difference. In all the figures, we observe that as the similarity gets higher, the execution time difference becomes smaller. In particular, we observe that the execution time differences of the target DB IO window and its top-3 most similar DB IO windows (i.e., points in the red box) are smallest.[5] We found the same trends in the results for the other target DB IO windows not shown in the figures.

### 5.3 Effectiveness of FORESEE

In this section, we verify the effectiveness of FORESEE. As mentioned in Section 5.2, the accuracy of the execution time estimated by FORESEE can be affected by the values of parameters (i.e., the number of similar IO windows ($p$) and the number of similar clusters ($l$)) used in matching. Thus, we first examine the sensitivity according to the parameters and show the accuracy of the execution times estimated by FORESEE. As an accuracy measure, we used a similarity-based metric called *Pearson correlation coefficient* (PCC); moreover, we also used another metric called *relative error* (RE) that measures the difference between the actual ($t_{act}$) and estimated ($t_{est}$) execution times as shown in Equation (6)

5. In Section 5.3.1, it is shown that the estimation with the top-3 most similar DB IO windows provides the best accuracy in our FORESEE framework.

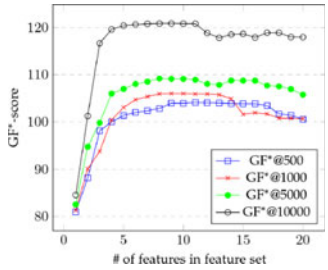



Fig. 8. GF* result.

$$RE(t_{est}, t_{act}) = \frac{|t_{est} - t_{act}|}{t_{act}}. \quad (8)$$

In order to show the effectiveness of FORESEE, we compared the accuracies of the estimated execution times of FORESEE and baseline methods. We searched for the alternative methods of estimating the execution time on SSDs, thereby finding the methods used in [7] and [8], termed MSST-11 and CIKM-2017, respectively, in our paper. Therefore, we used MSST-11 [7] and CIKM-2017 [8] as the primary competitors in our evaluation. In addition, references [39], [40] proposed the methods of estimating the execution time on HDDs; however, we did not employ them as our competitors because they are known to be inappropriate for the SSDs [7]. The MSST-2011 first extracts feature values from each DB IO window and constructs a regression tree based on them. It then estimates the execution time of the query IO window by inserting the feature values of the query IO window into the tree. The CIKM-2017 is our previous forecasting method proposed in [8]. Our current method in this paper employs 9 features, carefully selected by using our proposed goodness function$*$ as the final feature set, among total 20 features, while the previous method simply uses 3 features, which have been known to affect the SSD performance, without considering any selection criteria.

In addition to our two competitors (i.e., MSST-11 and CIKM-2017), we included the two trivial methods (i.e., RANDOM and AVG-PAGE-IO) in order to show the effectiveness of FORESEE by comparing it with the trivial methods that play a role of baselines. The RANDOM is a method for estimating the execution time of a query IO window. It estimates the execution time of a query IO window by randomly selecting a DB IO window from the DB and regarding its execution time as the execution time of the query IO window. The AVG-PAGE-IO first computes the average processing time of reading (resp. writing) 1 KB, the minimum unit (i.e., page) of IO requests. It then estimates the total execution time for reading (resp. writing) in an IO window by multiplying the total size of reading (resp. writing) in KB and the average processing time of reading (resp. writing) 1 KB. It finally estimates the total execution time of the IO window by summing the two execution times for total reading and total writing. These four baseline methods estimate the execution time of an entire query IO trace by adding the estimated execution times of all query IO windows extracted from the query IO trace.

For each baseline method, we measure the RE of the estimated execution time of every query IO trace (*resp.* query IO window) in comparison with its actual execution time and also measure the PCC of the estimated and actual

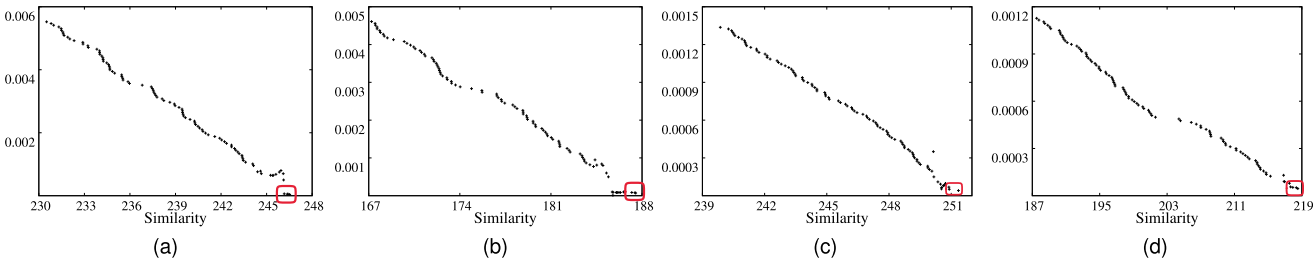



Fig. 9. Scatter plot of the actual and estimated execution times of query IO traces on the Hynix SSD.

execution times of all query IO traces (*resp.* query IO windows). We finally compute their average values as accuracies and compare the accuracies of all methods.

### 5.3.1 Parameter Sensitivity

In order to examine the sensitivity to parameters ($p$, $l$), we estimate the execution times of query IO windows with different parameter values. We vary $p$ 1 to 25 while $l$ is fixed to 1. Similarly, we vary $l$ 1 to 20 while $p$ is fixed to 1. For these experiments, we select 10,000 query IO windows randomly among all 82,847 query IO windows and estimate the execution times of those query IO windows selected. Finally, we measure the average of the RE for the accuracy.

Fig. 10a shows the RE of estimating the execution times of query IO windows with different numbers of similar IO windows ($p$). The $x$-axis indicates the value of $p$ and the $y$-axis indicates the RE. When $1 \leq p \leq 3$, the values of the RE are similar and the lowest RE (1.06) is obtained when $p$ is 1; when $p \geq 4$, as $p$ increases, RE gradually increases. This is because, with a large value of $p$, the DB IO window(s) less similar to the query IO window could be used in execution time estimation. As a result, the estimation accuracy would decrease in this case.

Fig. 10b shows the RE of estimating the execution times of query IO windows with different numbers of clusters ($l$). The $x$-axis indicates the value of $l$ and the $y$-axis indicates the RE. When $1 \leq l \leq 6$, the values of the RE decrease as $l$ increases and the lowest RE (0.69) is obtained when $l$ is 6; this is because it is more likely to find a DB IO window that is most similar to a query IO window if we examine more clusters. When $l \geq 6$, the values of the RE are not changed as $l$ increases; this is because a DB IO window that is most similar to a query IO window is already found when $l \leq 6$. In the following experiments, we set $p$ to 3 and $l$ to 6.

### 5.3.2 Estimation Accuracy

To verify the effectiveness of FORESEE, we estimated the execution time of 842,847 query IO windows. Fig. 11 shows the estimation accuracy of FORESEE and the four baseline methods by the scatter plot of the actual and estimated execution times of query IO windows. The $x$-axis indicates the estimated execution time and the $y$-axis indicates the actual execution time. A point denotes the coordinates of estimated and actual execution times of an IO window. In Fig. 11a, most points are located in the diagonal line, which indicates that FORESEE accurately estimates the execution times of query IO windows. In Figs. 11b and 11c, most points are randomly distributed, which indicates that the RANDOM and the AVG-PAGE-IO methods do not work well in estimating the execution times of query IO windows. Finally, in Figs. 11d and 11e, most points are located in the diagonal line, which indicates that the CIKM-2017 and the MSST-2011 estimate the execution times of query IO windows more accurately than the other baseline methods. Table 6 shows that the PCC and the RE of FORESEE and the four baseline methods. We observe that FORESEE outperforms the baseline methods in terms of both PCC and RE (i.e., having higher PCC and lower RE).

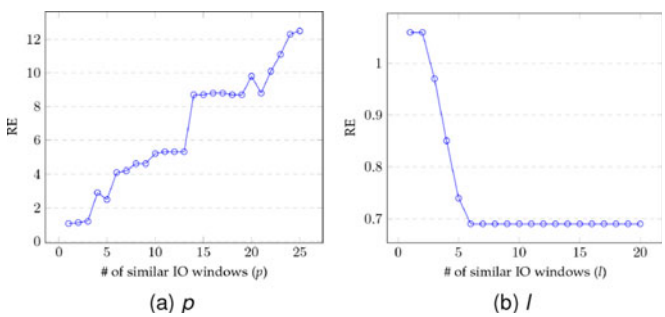



Fig. 10. The accuracy of FORESEE with different parameter values.

We estimated the execution time of 101 query IO traces. Fig. 12 shows the estimation accuracy of FORESEE and the four baseline methods on the Hynix SSD by the scatter plot of the actual and estimated execution times of IO traces. The $x$-axis indicates the estimated execution time and the $y$-axis indicates the actual execution time. A point denotes the coordinates of the estimated and actual execution times of an IO trace. In Fig. 12a, most points are located in the diagonal line, which indicates that FORESEE accurately estimates the execution times of the query IO traces. In Figs. 12b and 12c, most points are randomly distributed, which indicates that the RANDOM and the AVG-PAGE-IO methods do not work well in estimating the execution times of query IO traces. Finally, in Figs. 12d and 12e, most points are located in the diagonal line, showing that its estimated execution times of query IO traces are reasonably accurate, compared with the other baseline methods. However, Table 7 shows that FORESEE

TABLE 6
Accuracies of Execution Time Estimation of IO Windows on Various SSDs

| | Hynix | | Samsung | | Intel | | Plextor | |
|---|---|---|---|---|---|---|---|---|
| | PCC | RE | PCC | RE | PCC | RE | PCC | RE |
| FORESEE | 0.89 | 1.04 | 0.86 | 1.12 | 0.90 | 1.05 | 0.88 | 1.04 |
| **RANDOM** | 0.04 | 6.47 | 0.23 | 6.23 | 0.06 | 8.12 | 0.26 | 5.24 |
| **AVG-PAGE-IO** | 0.16 | 5.39 | 0.12 | 7.12 | 0.25 | 6.81 | 0.23 | 5.46 |
| **CIKM-2017** | 0.71 | 1.28 | 0.76 | 1.51 | 0.78 | 1.34 | 0.70 | 1.39 |
| **MSST-2011** | 0.69 | 1.35 | 0.77 | 1.46 | 0.75 | 1.51 | 0.67 | 1.48 |

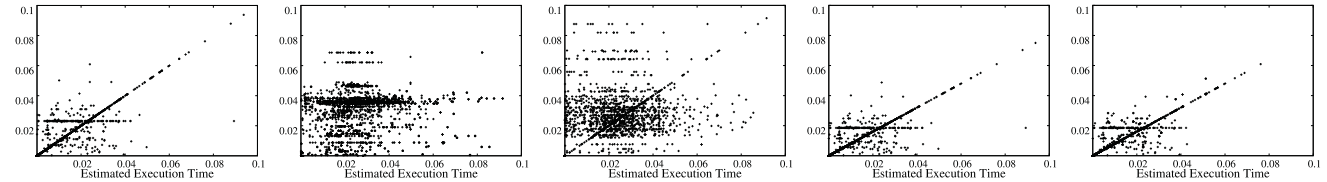



Fig. 11. Scatter plot of the actual and estimated execution times of query IO windows.

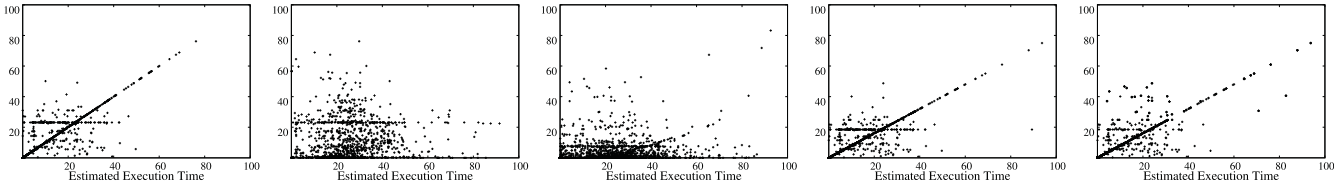



Fig. 12. Scatter plot of the actual and estimated execution times of query IO traces on the Hynix SSD.

outperforms the four baseline methods including the MSST-2011 significantly in terms of both PCC and RE regardless of SSDs.

## 6 Conclusion

In this paper, we propose a novel framework named FORESEE for estimating the execution time of a query IO trace on a given SSD. In the literature, there is an important observation that if two IO traces are similar to each other in their IO behavior, their execution times tend to be similar when they run on the same SSD. Based on this observation, FORESEE estimates the execution time of a query IO trace by using the execution time of DB IO traces similar to the query IO trace. To solve the problems caused by different lengths of IO traces, we define a new concept called an *IO window* with a fixed length. FORESEE first estimates the execution times of the IO windows extracted from the query IO trace and then estimates the execution time of the query IO trace by aggregating the estimated execution times of the query IO windows. This approach of FORESEE makes it possible to estimate accurately the execution time of a query IO trace of an arbitrary length. Through the extensive experiments, we show that FORESEE can estimate the execution time of query IO windows and query IO traces very accurately.

TABLE 7
Accuracies of Execution Time Estimation of IO Traces on Various SSDs

| | Hynix | | Samsung | | Intel | | Plextor | |
|---|---|---|---|---|---|---|---|---|
| | PCC | RE | PCC | RE | PCC | RE | PCC | RE |
| FORESEE | 0.87 | 1.07 | 0.84 | 1.16 | 0.87 | 1.06 | 0.87 | 1.05 |
| **RANDOM** | 0.14 | 8.45 | 0.21 | 6.46 | 0.04 | 8.30 | 0.21 | 5.87 |
| **AVG-PAGE-IO** | 0.12 | 9.63 | 0.07 | 7.41 | 0.21 | 7.10 | 0.19 | 5.90 |
| **CIKM-2017** | 0.81 | 1.31 | 0.74 | 1.59 | 0.81 | 1.67 | 0.68 | 1.41 |
| **MSST-2011** | 0.79 | 1.39 | 0.74 | 1.60 | 0.77 | 1.83 | 0.63 | 1.52 |

## Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT; MSIT) under Grant NRF-2020R1A2B5B03001960, in part by the Next-Generation Information Computing Development Program through the NRF funded by the MSIT under Grant NRF-2017M3C4A7083678), and in part by NRF grant funded by the MSIT under Grant 2018R1A5A7059549.

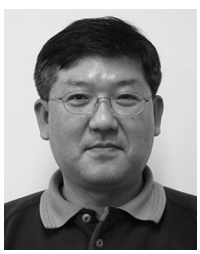
## References

[1] S. Lee *et al.*, "A case for flash memory SSD in enterprise database applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1075–1086.
[2] S. Lee, B. Moon, and C. Park, "Advances in flash memory SSD technology for enterprise database applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 863–870.
[3] J. Yoon and G. Tressler, "Advanced flash technology status, scaling trends and implications to enterprise SSD technology enablement," in *in Proc. Flash Memory Summit*, 2012, pp. 1–16.
[4] L. Bouganim, B. Jónsson, and P. Bonnet, "uFLIP: Understanding flash IO patterns," in *Proc. Biennial Conf. Innovative Data Syst.*, 2009, pp. 1–12.
[5] A. TraegerandE. Zadok,"A nine year study of file system and storage benchmarking," *ACM Trans. Storage*, vol. 4, no. 2, pp. 1–56, 2008.
[6] Y. Lu, J. Shu, andJ. Zhang,"Mitigating synchronous I/O overhead in file systems on open-channel SSDs," *ACM Trans. Storage*, vol. 15, no. 3, pp. 17–52, 2007.
[7] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," in *Proc. IEEE Int. Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–11.
[8] Y. Kang *et al.*, "A framework for estimating execution times of IO traces on SSDs," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2017, pp. 2123–2126.
[9] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2011.
[10] N. Agrawal, V. Prabhakaran, and T. Wobber, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.
[11] D. Bae *et al.*, "Intelligent SSD: A turbo for big data mining," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2013, pp. 1573–1576.
[12] S. Kim *et al.*, "Fast, energy efficient scan inside flash memory SSD," in *Proc. Int. Workshop Accelerating Data Manage. Syst. Using Modern Processor Storage Archit.*, 2011, pp. 1–8.

[13] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE Int. Conf. High Perform. Comput. Archit.*, 2011, pp. 266–277.
[14] J. Shin *et al.*, "FTL design exploration in reconfigurable high-performance SSD for server applications," in *Proc. ACM Int. Conf. Supercomput.*, 2009, pp. 338–349.
[15] G. Goodson and R. Iyer, "Design tradeoffs in a flash translation layer," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. Workshop Use Emerg. Storage Memory Technol.*, 2010, pp. 1–7.
[16] S. Leeetal, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, pp. 1–27, 2007.
[17] T. Chung *et al.*, "System software for flash memory: A survey," in *Proc. Int. Conf. Embedded Ubiquitous Comput.*, 2006, pp. 394–404.
[18] S. Leeetal, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *ACM Trans. Special Interest Group Operating Syst.*, vol. 42, no. 6, pp. 36–42, 2008.
[19] Y. Hu *et al.*, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. ACM Int. Conf. Supercomput.*, 2011, pp. 96–107.
[20] X. Hu and R. Haas, "The fundamental limit of flash random write performance: Understanding, analysis and performance modelling," IBM Res. Rep. RZ 3771, pp. 1–15, 2010.
[21] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multi-level parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.
[22] H. Hwang and Y. Yu, "An analytical method for estimating interpreting query time," in *Proc. Int. Conf. Very Large Database Endowment*, 1987, pp. 347–358.
[23] C. Dirik and B. Jacob, "The performance of PC solid-state-disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proc. ACM Int. Symp. Comput. Archit.*, 2009, pp. 279–289.
[24] R. KollerandR. Rangaswamii,"I/O deduplication: Utilizing content similarity to improve I/O performance," *ACM Trans. Storage*, vol. 6, no. 3, pp. 1–13, 2010.
[25] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1994, pp. 419–429.
[26] Y.-S. Moon, K.-Y. Whang, and W.-K. Loh, "Duality-based subsequence matching in time-series databases," in *Proc. IEEE Int. Conf. Data Eng.*, 2001, pp. 263–272.
[27] B. Seo, S. Kang, J. Choi, J. Cha, Y. Won, and S. Yoon, "IO workload characterization revisited: A data-mining approach," *IEEE Trans. Comput.*, vol. 63, no. 12, pp. 3026–3038, Dec. 2014.
[28] M. Calzarossa, L. Massari, andD. Tessera,"Workload characterization: A survey revisited," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–43, 2016.
[29] J. Bang *et al.*, "HPC workload characterization using feature selection and clustering," in *Proc. ACM Int. Workshop Syst. Netw. Telemetry Analytics*, 2020, pp. 33–40.
[30] T. Corman *et al.*, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.
[31] V. Ganti, J. Gehrke, and R. Ramakrishnan, "Mining very large databases," *IEEE Comput.*, vol. 32, no. 8, pp. 38–45, Aug. 1999.
[32] S. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1993, pp. 22–31.
[33] UMass Trace Repository, Search Engine I/O, 2007. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage
[34] Y. Hu *et al.*, "Avoiding the streetlight effect: I/O workload analysis with SSDs in mind," in *Proc. USENIX Conf. Hot Topics Storage File Syst.*, 2016, pp. 36–40.
[35] I. Picoli *et al.*, "uFLIP-OC: Understanding flash I/O patterns on open-channel solid-state drives," in *Proc. Asia-Pacific Workshop Syst.*, 2017, pp. 1–7.
[36] F. Wan, G. Han, G. Jia, J. Wan, and C. Jiang, "Application-grained block I/O analysis for edge computational intelligent," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun. IEEE 17th Int. Conf. Smart City IEEE 5th Int. Conf. Data Sci. Syst.*, 2019, pp. 381–388.
[37] G. Prasadd, A. Cheung, and D. Suciu, "Handling highly contended OLTP workloads using fast dynamic partitioning," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 527–542.
[38] A. Kamsky, "Adapting TPC-C benchmark to measure performance of multi-document transactions in MongoDB," in *Proc. Int. Conf. Very Large Data Bases*, 2019, pp. 2254–2262.
[39] M. Uysal, G. A. Alvarez, and A. Merchant, "A modular, analytical throughput model for modern disk arrays," in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2001, pp. 183–192.
[40] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger, "Storage device performance prediction with CART models," in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2004, pp. 588–595.

**Yoonsuk Kang** received the BS degree from Hanyang University, Seoul, South Korea, in 2013. He is currently working toward the PhD degree in computer science at Hanyang University, Seoul, South Korea. His current research interests include data mining, database, and social network analysis.

**Yong-Yeon Jo** received the PhD degree in computer science from Hanyang University, Seoul, South Korea, in 2018. He is currently a researcher with Medical AI Company, Ltd in Korea. His research interests include healthcare, biosignal, electrocardiogram, deep learning, and data mining.



**Jaehyuk Cha** received the BS, MS, and PhD degrees in computer science, all from Seoul National University, Seoul, South Korea, in 1987, 1991, and 1997, respectively. He was with the Korea Research Information Center from 1997 to 1998. Since 1998, he has been with the Division of Computer Science, Hanyang University, Seoul, South Korea, as a professor. His research interests include flash storage system, technology-enhanced learning, and computational social science.

**Wan D. Bae** received the BS degree in architectural engineering from Yonsei University, Seoul, South Korea, in 1989, and the MS and PhD degrees in computer science from the University of Denver, Denver, Colorado, in June 2004 and November 2007, respectively. She is currently an associate professor with the Department of Computer Science, Seattle University. She authored two book chapters and more than 30 research papers in the areas of databases, GIS, and health informatics. Her current research interests include spatial and spatio-temporal data mining, health informatics, mobile computing and optimization, big data analytics, and GIS.

**Wonjun Lee** received the BS and MS degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1989 and 1991, respectively, the MS degree in computer science from the University of Maryland, College Park, Maryland, in 1996, and the PhD degree in computer science and engineering from the University of Minnesota, Minneapolis, Minnesota, in 1999. In 2002, he joined the Faculty of Korea University, Seoul, where he is currently a professor with the School of Cybersecurity. His research interests include communication and network protocols, optimization techniques in wireless communication and networking, security and privacy in mobile computing, and data center networking.

**Sang-Wook Kim** (Member, IEEE) received the BS degree in computer engineering from Seoul National University, Seoul, South Korea, in 1989, and the MS and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 1991 and 1994, respectively. From 1995 to 2003, he served as an associate professor with Kangwon National University. In 2003, he joined Hanyang University, Seoul, South Korea, where he is currently a professor with the Department of Computer Science and the director of the Brain-Korea-21-FOUR research program. He is also leading a National Research Lab (NRL) Project funded by the National Research Foundation, since 2015. From 2009 to 2010, he visited the Computer Science Department, Carnegie Mellon University, as a visiting professor. From 1999 to 2000, he worked with IBM T. J. Watson Research Center, New York, as a postdoc. He also visited the Computer Science Department, Stanford University as a visiting researcher, in 1991. He is an author of more than 200 papers in refereed international journals and international conference proceedings. His research interests include databases, data mining, multimedia information retrieval, social network analysis, recommendation, and web data analysis. He is a member of ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.