IO Workload Characterization Revisited: A Data-Mining Approach

Bumjoon Seo, Sooyong Kang, Jongmoo Choi, Jaehyuk Cha, Youjip Won, and Sungroh Yoon, *Senior Member*, *IEEE*

Abstract—Over the past few decades, IO workload characterization has been a critical issue for operating system and storage community. Even so, the issue still deserves investigation because of the continued introduction of novel storage devices such as solid-state drives (SSDs), which have different characteristics from traditional hard disks. We propose novel IO workload characterization and classification schemes, aiming at addressing three major issues: (i) deciding right mining algorithms for IO traffic analysis, (ii) determining a feature set to properly characterize IO workloads, and (iii) defining essential IO traffic classes state-of-the-art storage devices can exploit in their internal management. The proposed characterization scheme extracts basic attributes that can effectively represent the characteristics of IO workloads and, based on the attributes, finds representative access patterns in general workloads using various clustering algorithms. The proposed classification scheme finds a small number of representative patterns of a given workload that can be exploited for optimization either in the storage stack of the operating system or inside the storage device.

Index Terms—IO workload characterization, storage and operating systems, SSD, clustering, classification

1 Introduction

As of 2013, NAND flash prices are finally below one US dollar per gigabyte [1], [2], and NAND-based SSDs are quickly gaining momentum as the primary storage device in mobile platforms, notebooks, and even enterprise servers [3]-[5]. In addition to the fact that it has no mechanical parts, NAND flash memory uniquely distinguishes itself from dynamic random access memory (DRAM) and hard disk drives (HDDs) in several key aspects: (i) the asymmetry in read and write latency, (ii) the existence of erase operations (i.e., inability to overwrite), and (iii) a limited number of erase/write cycles. NAND-based SSDs harbor a software entity called the *flash translation layer* (FTL) for hiding the shortcomings of the underlying NAND device and exporting a logical array of blocks to the file system. An FTL algorithm should be carefully crafted to maximize IO performance (latency and bandwidth) and to lengthen the NAND-cell life time.

The efficient design of SSD hardware (e.g., internal parallelism and buffer size), and software (e.g., address mapping [6]-[10], wear leveling [11]-[13] and garbage collection algorithm [13], [14]) mandates solid understanding of underlying IO workloads. However, existing characterization efforts for HDDs and virtual memory fail to meet the characterization requirements for NAND-based storage devices. Over the past few decades, IO characterization has been under intense research in the operating and storage systems community, reaching sufficient maturity as far as HDD-based storage systems are concerned [15]-[19].

Engineers have carefully chosen the attributes for the characterization properly incorporating the physical characteristics of the hard disk. For HDDs, the spatial characteristics of workloads are of primary concern since the head movement overhead constitutes the dominant fraction of IO latency and performance. The IO characterization studies for HDDs thus have focused on determining the spatial characteristics (e.g., randomness) of block-access sequences. For memory subsystems (e.g., buffer cache and virtual memory), it is critical to characterize the temporal aspects (e.g., a working set) of memory access, because this determines the cache-miss behavior.

NAND-based SSDs further lead us to consider the notion of IO characteristics in three-dimensional space: spatial characteristics (i.e., access locations: random versus sequential), temporal characteristics (i.e., access frequency of given data: hot versus cold), and volume characteristics (i.e., size of access: single versus multiple blocks). Based on the characteristics of incoming workloads, the SSD controller can tune the parameters for key housekeeping tasks such as log-block management [20], mapping granularity decision [21], and victim selection for garbage collection [13], [14].

Some approaches proposed to examine the contents of IO when analyzing IO workloads (e.g., for implementing deduplication in an SSD controller [22]-[24] and for designing caches [25]-[27]). While the value locality can be widely used for determining the caching and deduplication schemes, the 'value' itself is highly context-dependent. In this work,

- *B. Seo was with the Department of Electrical and Computer Engineering, Seoul National University, Seoul 151-744, Korea, and is with the Emerging Technology Laboratory, Samsung SDS Co., LTD., Seoul 135-280, Korea.*
- *S. Kang, J. Cha, and Y. Won are with the Department of Computer Science and Engineering, Hanyang University, Seoul 133-791, Korea.*
- *J. Choi is with the Department of Software, Dankook University, Yongin 448-701, Korea.*
- *S. Yoon is with the Department of Electrical and Computer Engineering, Seoul National University, Gwanak-gu, Seoul 151-744, Korea. E-mail: sryoon@snu.ac.kr.*

Manuscript received 01 Mar. 2013; revised 23 Aug. 2013; accepted 02 Sep. 2013. Date of publication 12 Sep. 2013; date of current version 12 Nov. 2014. Recommended for acceptance by D. Talia.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.187

we thus focus on a generic framework to classify the workload and include only spatial, temporal and volume aspects of IO which can be analyzed without actually examining the contents of IO. In the cases in which the content examination is feasible, considering content locality in workload characterization and classification may produce improved results.

The advancement of NAND-based storage devices, which bear different physical characteristics from hard-disk-based storage devices, calls for an entirely new way of characterizing IO workloads. In this paper, we revisit the issue of understanding and identifying the essential constituents of modern IO workloads from the viewpoint of the emerging NAND-based storage device. In particular, we aim at addressing the following specific questions: (i) What is the right clustering algorithm for IO workload characterization's sake? (ii) What is the minimal set of features to effectively characterize the IO workload? and (iii) How should we verify the effectiveness of the established classes? These issues are tightly associated with each other and should be addressed as a whole. We apply a data-mining approach for this purpose. We perform a fairly extensive study for the comprehensiveness of the result: We examine eleven clustering algorithms and twenty workload features with tens of thousands of IO traces. We apply both unsupervised and supervised learning approaches in a streamlined manner. In particular, we apply unsupervised learning first so that existing domain knowledge does not bring any bias on establishing a set of traffic classes.

For SSDs, we can apply the proposed approach to intelligent log-block management in hybrid mapping, victim selection in garbage collection, and mapping-granularity decision in multi-granularity mapping, although this paper does not present the details of such applications. We further believe that the proposed methodology can be applied to other types of storage devices for IO optimization.

The rest of this paper is organized as follows: Section 2 presents prior work related to this paper, and Section 3 briefly describes the overview of our work. Section 4 defines parameters and features for IO workload characterization. In Section 5, we select the most effective three features among a total of twenty features and extract representative IO patterns using unsupervised learning techniques. In Sections 6 and 7, we present a tree-based model of IO traces and a decision-tree-based pattern classification scheme, respectively. We conclude the paper in Section 8.

2 Background

2.1 Related Work

Data mining has been used for analyzing and evaluating various aspects of computer systems. Lakhina et al. used entropy-based unsupervised learning to detect anomalies in network traffic [28]. Won et al. applied an AdaBoost technique [29] to classify soft real-time IO traffic [30]. Park et al. employed Plackett-Burgman designs to extract the principle features in file system benchmarks [31].

Regarding IO workload characterization, there have been empirical studies on characterizing file system access patterns and activity traces [32], [33]-[35]. Application-specific workload characterization efforts exist for web servers [36], [37], decision support systems [38], and personal computers [39]. These approaches draw simple statistics on a limited set of attributes (e.g., read/write ratios, IO size distribution, randomness of IO), thus often failing to properly characterize the correlations among different attributes.

The workload analysis and classification can be used at different layers of the IO stack. Specifically, Li et al. proposed to use a data-mining algorithm at the block device driver for disk scheduling, data layout and prefetching [40]. Sivathanu et al. proposed to embed the analysis function inside the file system layer, which allows inference of the location of metadata region of the file system via the examination of the workload access pattern [41]. Won et al. proposed to embed the analysis function at the firmware of the storage device [30] and aimed at exploiting the workload characteristics to control the internal behavior of the storage device (e.g., block allocation, the number of retries and error-correction coding).

Sophisticated workload generators and measurement tools have been proposed for devising pattern-rich microbenchmarks [42]-[46]. These tools are useful for producing arbitrary workload patterns in a fairly comprehensive manner. However, without the knowledge on essential patterns in real-world workloads, it is not easy to generate appropriate IO workloads for accurate performance measurement.

The most important limitation out of all previous approaches to IO workload characterization may be that these approaches considered only HDDs as the storage device. The workload characterization for rotating media is relatively simple, because the dominant factor regarding the device performance is the seek overhead. Hence, characterizing workloads in terms of sequentiality and randomness has provided sufficient, even though not complete, ingredients to evaluate the performance of the file and storage stack.

Now that the NAND-based SSD is emerging as a strong candidate for the replacement of the HDD, it is necessary to characterize workloads in terms of more abundant attributes to accurately measure the performance of the file and storage stack. Recently, a workload generator called uFlip [47], [48] was proposed, which can generate more diverse workloads tailored for examining the performance of NAND-based SSDs. However, the essential IO patterns used in uFlip were not derived from real-world workloads. Thus, it is not known whether these patterns are sound and complete ones for SSDs.

The objective of this work is to find the presentative patterns of IO workloads from NAND storage's point of view. These patterns are to be exploited in designing various components of NAND-based storage device (e.g., address mapping, wear leveling, garbage collection, write buffer management, and other key ingredients).

2.2 Flash Translation Layer (FTL)

NAND flash memory has unique characteristics in contrast to DRAM and HDDs. In particular, the flash translation layer (FTL) [6]-[10] plays a key role in operating NAND-based SSDs. The primary objective of FTL is to let existing software recognize an SSD as a legacy, an HDD-like block device. FTL performs three main tasks: (i) FTL maintains a logical-address to physical-address translation table (i.e., address mapping), (ii) FTL consolidates valid pages into a block so that invalid pages can be reset and reused (i.e., garbage collection), and (iii) FTL evens out the wears of blocks by properly selecting physical blocks (i.e., wear leveling). To maximize IO performance (latency and bandwidth) and to lengthen the NAND-cell life time, FTL designers should carefully craft the algorithms for selecting destination blocks for writing, for determining victim blocks for page consolidation, and for determining when to perform such consolidation.

The success of FTL lies on properly identifying incoming traffic and effectively optimizing its behavior. As a result, there have been various approaches to exploiting spatial, temporal and volume aspects of IO workloads in the context of FTL algorithms. By allocating separate log blocks for sequential and random workloads and managing them differently, we can significantly reduce the overhead for merging log blocks in hybrid mapping [6]. According to [7], we can significantly improve the log-block utilization and overall system performance by maintaining separate log-block regions based upon the hotness of data. It was reported that we can reduce the mapping table size by adopting different mapping granularities, thus exploiting the IO-size distribution of incoming workloads [49]. Although these methods are pioneering, they show rather unsatisfactory performance in classifying workloads, typically producing frequent misclassification. We can conclude that there is a clear need for a more elaborate IO characterization framework for advancing SSD technology even further.

2.3 Applications of Workload Analysis to SSD

We can utilize IO workload analysis results in multiple ways for optimizing the performance of NAND-based SSDs. We present a few ideas here: enhancing write performance, reducing FTL mapping information, and improving garbage collection. The actual implementation of such ideas will be part of our future work.

As elaborated in Section 4.2, we view and collect IO traces in the unit of the size of a write buffer, since this size corresponds to the largest size that an FTL can directly manage for efficiently performing flash write operations. If certain write patterns are found to be favorable for fast write operations (e.g., fully sequential patterns), then the FTL can reorder and merge the IO data pages stored temporarily in the write buffer, transforming them into one of the patterns that can improve write performance.

Furthermore, we can significantly reduce the amount of FTL mapping information by workload pattern analysis. Given that the capacity of SSDs is ever growing, using the conventional page-mapping scheme for terabyte-scale SSDs would require gigabyte-scale mapping information. We could use coarse-grained mapping, such as the block mapping, but this would significantly degrade the random-write performance. For the data incurring few random writes (e.g., multimedia data), we had better use coarse-grained mapping, and for the data incurring frequent random writes (e.g., file system metadata), we had better rely on fine-grained mapping, such as page mapping. If an FTL can distinguish and classify different data types automatically, then it can determine the optimal mapping granularity for each data type and minimize the amount of mapping information while minimizing the decrease in IO performance.

Additionally, we can enhance the performance of garbage collection by applying the patterns discovered by IO workload analysis to data distribution. Inside recent SSDs, the NAND flash chips are typically configured in multiple channels, each of which consists of multiple ways. To exploit such a multi-channel/multi-way architecture, SSD controllers distribute write requests over multiple channels and ways. This can scatter data with locality over separate chips. If these data are later modified, then the invalid data end up existing in multiple chips, which significantly deteriorates garbage-collection performance. To alleviate this issue, we need to keep data with locality in the same chip, to also exploit way interleaving.

One solution is to utilize the sequential random (SR) pattern discovered by our approach (see Table 3 and Fig. 6). We write sequential IOs to a single chip, thus maintaining locality, and write random IOs to remaining chips in parallel, thus achieving the effect of way interleaving. Going one step further, we can divide sequential IOs into NAND-block-sized chunks and then write them to different chips in parallel. This will allow us to achieve interleaving effects, while still maintaining locality (recall that the unit of erase is one NAND block). This type of data distribution can make valid and invalid pages exist in different blocks by the effect of maintaining locality and eventually deliver positive effects on garbage collection.

3 Overview

Fig. 1 shows the overall flow of our approach. Our objective is to find a set of representative patterns that can characterize IO workloads, and to propose a model that can later be used to classify IO traces for class-optimized processing. In this paper, we first focus on defining such patterns of IO traces and constructing a model based on those patterns (depicted inside the dot-ted box labeled *DESIGN TIME* in Fig. 1). Then, we show how to construct a simple but accurate classifier based on this model. We can include this classifier in the firmware of a storage device and use it in runtime (depicted inside the box labeled *RUNTIME*).

We first collect a number of IO trace samples and represent each of them mathematically as a vector of various features. Then, in the first phase of our approach, we analyze the vectorized IO traces by clustering. Our hope is that, by applying an unsupervised learning approach, we can find novel patterns of IO traces in addition to re-discovering known ones. We refer to the resulting clusters from this first phase as *preliminary IO trace patterns*.

Based on the clustering result from the previous step, we build a tree-based model that can represent characteristics of IO traces. Although clustering can give us an idea of what types of patterns appear in today's IO workloads, we find that some of the preliminary patterns are redundant or have little practical value. We streamline the preliminary clusters and define a new set of IO classes. We then represent them by a tree structure, due to the hierarchical nature of the classes of IO patterns.

We can apply the proposed model and classes of patterns to classify incoming workloads efficiently. We can optimize various SSD controller algorithms for each class of the IO workloads so that the controller dynamically adapts its behavior subject to the characteristics of incoming IO

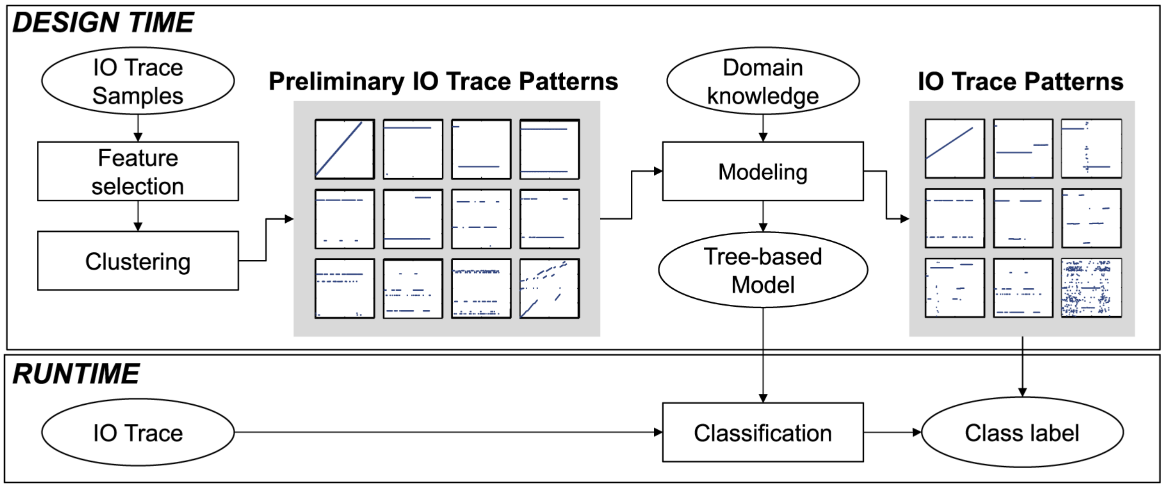

Fig. 1. Overview of proposed approach.

(e.g., random vs. sequential, hot vs. cold, degree of correlations among IO accesses, garbage collection algorithms [13], [14], log block merge algorithms [6]-[9], and wear leveling [11]-[13]).

4 Characterizing IO Traces

4.1 Parameter Definitions

An IO trace consists of multiple IO requests, each of which includes information on the logical block address (LBA) of the starting page, access type (read or write), and the number of pages to read or write. Fig. 2a shows an example trace that consists of five IO requests. As shown in Fig. 2b, we can also represent an IO trace as a series of LBA over pages. We can denote an IO trace by function $f : P \rightarrow L$, where P is a set of page indices and L a set of LBAs. We assume that P or L is a set of nonnegative integers.

We can characterize an IO trace using the following parameters and definitions:

Page size: we assume that the unit of a read or write is a page and that the size of each page is equal (e.g., 4 KB) for all traces.

Trace size: the size of a trace is the total number of unique pages in the trace, or $|P|$.

Segment: a segment is a continuous LBA sequence in an IO trace. Fig. 2b shows four segments. Note that multiple IO requests can form a single segment. For example, segment 1 in Fig. 2b consists of two IO requests: RQ1 and RQ2.

Break point (BP): a BP is the end of a segment. A BP occurs at page $p_1 \in P$ if $f(p_1 + 1) \neq f(p_1) + 1$. In Fig. 2b, we indicate each BP by a solid circle. By definition, the number of BPs is identical to that of segments.

Continued point (CP): a CP is the start of a segment that continues another earlier segment in the same trace. If the break point of a segment occurs at $p_1 \in P$, then $p_2 \in P$ is a CP if $f(p_2) = f(p_1) + 1$ and $p_2 - p_1 > 1$. For example, the start point of segment 3 in Fig. 2b is a CP. The start points of the other segments are not.

Access length: this is the number of pages (or bytes) each IO request causes the IO device to read or write. For a segment that consists of multiple IO requests, the segment length is the sum of the access lengths of the component IO requests.

Virtual segment: this comprises two or more segments that can form a continuous segment if we remove the intervening pages between them. Two segments s and t form a virtual segment if $f(p_t) = f(p_s) + 1$ and $p_t - p_s > 1$, where p_s and p_t are the last and first pages of s and t, respectively. In Fig. 2b, segments 1 and 3 form a virtual segment.

Random segment: if a segment is smaller than a user-specified threshold and is not part of any virtual segment, then we call this segment *random*. In Fig. 2b, if the user-specified threshold happens to be twice the length of segment 4, then segment 2 is not random but segment 4 is.

Up- and down-segment: if the start LBA of a segment is greater than the BP of the previous segment, then we call this segment an up-segment. (A down-segment is the opposite.) In Fig. 2b, segments 2 and 4 are up-segments, whereas segment 3 is a down-segment.

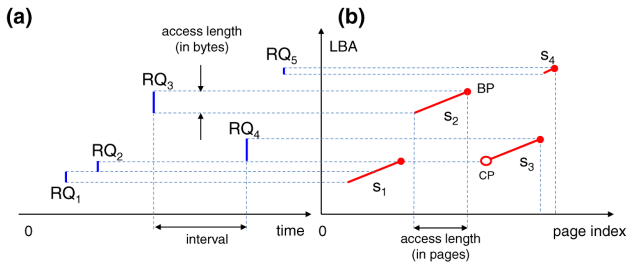

Fig. 2. Parameter definitions for characterizing IO traces.

4.2 IO Trace Data Collection

We utilize a public data set [50], which consists of workloads from two web servers (one hosting the course management system at the CS department of Florida International University, USA, and the other hosting the department's web-based email access portal), an email server (the user 'inboxes' for the entire CS department at FIU) and a file server (an NFS server serving home directories of small research groups doing software development, testing, and experiments).

In addition, we collect IO trace data from a system that has the following specifications: Intel Core 2 Duo E6750 CPU, 2 GB 400 MHz DDR2 DRAM, Linux 2.6.35-28-generic, and a

500 GB Seagate ST3500320AS HDD. To generate stride read and write traces, we use IOzone version 3.308. We use the blktrace tool to collect IO traces during Windows 7 installations, text processing via MS Word version 2007, web surfing using Google Chrome and MS Explorer and downloading multimedia files.

The write buffer size of the SSD controller governs the size of the observation window for classification. The length of the IO trace in the clustering should be tailored to properly incorporate the size of the write buffer. In our study, we presume that the write buffer size is 32 MByte [51], [52], which can hold 8,000 unique 4-KByte pages. This sets the length of a trace to 8,000. In total, we generated 58,758 trace samples, each of which has 8,000 unique pages.

To implement the IO trace analysis and data-mining algorithms described in this paper, we use the R2010 b version of the MATLAB programming environment (http://www.mathworks.com) under 64-bit Windows 7.

4.3 Feature Definitions

Based on the above definitions, we characterize each IO trace by the twenty features listed in Table 1. Features F1 and F2 indicate the length of a trace in terms of pages and time. Feature F3 is the length of the longest segment. Features F4-F7 represent the total number of IO requests, BPs, CPs and non-random segments. The other features are the ratios (F8-F10) and basic statistics, such as arithmetic means (F11-F13), range (F14), and standard deviations (F16-F20) of other features. Using these features, we transform each IO trace sample into a vector of twenty dimensions for downstream analysis. That is, we convert the input trace data into a matrix of 58,758 rows (recall that we used 58,758 traces in our experiments) and twenty columns and then normalize this matrix so that every column has zero mean and unit variance.

We consider various features hoping that we can discover an important feature by exploring a variety of (combinations of) features, given that we do not know the best set of features for optimal IO characterization. However, due to the curse of dimensionality [53] (or lack of data separation in high-dimensional space), using too many features for data mining could make it difficult to separate IO traces in the original feature space. Thus, we need to select a set of features for better clustering results. In the next section, we will explain how to select a good set of features that yield a reasonable clustering result.

5 Finding Representative Patterns from IO Traces by Clustering

Before building a model for classifying IO traces, we first want to see what types of IO traces appear in storage systems. The exact number of all possible patterns may be difficult to determine, but we expect that we can find a small number of representative patterns. To summarize the patterns of IO traces in an automated fashion, we need a computational analysis tool that can handle a number of IO trace samples and find patterns therefrom. To this end, clustering is a perfect fit, since clustering (as unsupervised learning) enables us to find novel, previously unnoticed patterns beside known or expected ones. Classification is useful when the set of possible classes is already determined, but as a supervised learning technique, it cannot discover novel patterns, unlike clustering.

TABLE 1
List of Features for Characterizing IO Traces

ID	Feature description	a	b
F1	Trace size in pages		
F2	Duration of trace (= sum of all intervals between IO requests)		
F3	Length of the longest segment		○
F4	Number of IO requests		
F5	Number of BPs (= segments)		○
F6	Number of CPs		
F7	Number of non-random segments	○	○
F8	Ratio of the number of pages in random segments to that of all pages		○
F9	Ratio of the number of CPs to that of BPs	○	○
F10	Ratio of the number of up-segments to that of all segments		
F11	Average segment length		
F12	Average interval between IO requests		
F13	Average access length of IO requests		
F14	Range of intervals between IO requests		
F15	STD dev. of access lengths		
F16	STD dev. of the starting LBAs of IO requests	○	
F17	STD dev. of the page indices of BPs		
F18	STD dev. of the page indices of CPs		
F19	STD dev. of the starting page indices of random segments		
F20	STD dev. of intervals between IO requests		

[a]The features selected by backward elimination.
[b]The features used by tree-based model.

For clustering analysis of IO traces, we need to determine the set of features and the specific clustering algorithm used. Since the set of best features differs by which clustering algorithm is used, we need to consider the two dimensions at the same time for the optimal result. However, optimizing all of these simultaneously will be difficult due to the huge size of search space. We thus determine the set of features by a feature selection procedure, with the clustering algorithm fixed. Then, with respect to the determined set of features, we find the best clustering algorithm.

5.1 Evaluation Metric

To assess different sets of features and clustering methods, we first need to establish a metric that can assess the quality of a clustering result. Then, we search for the scheme that maximizes this metric. In this paper, we use the silhouette value [54], a widely used metric for scoring clustering results.

For a clustering result, we can assign each clustered sample a silhouette value. Then, we can represent the whole clustering result by the median silhouette value of all clustered samples. More specifically, we define $s(i)$, the silhouette value of sample i, as follows [54]:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \tag{1}$$

where $a(i)$ represents the average distance of sample i with all other samples within the same cluster, and $b(i)$ denotes the lowest average distance to i of the other clusters. By definition, a silhouette value can range from -1 to $+1$. $+1$ represents the perfectly clustered sample, whereas -1 indicates the opposite.

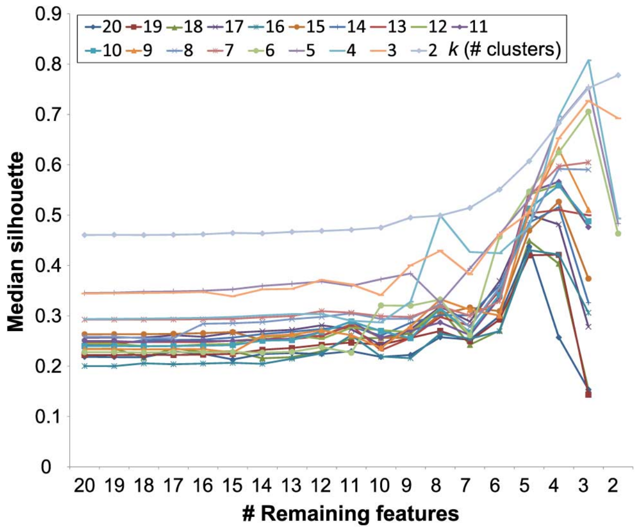

Fig. 3. Backward elimination process for feature selection (k denotes the number of clusters). The median silhouette value is maximized with $k = 4$ and three features.

5.2 Feature Selection

The feature selection refers to a procedure to determine a set of features that yield the best clustering result. The feature selection is computing intensive with the time complexity exponential to the number of features. Among existing heuristic approaches, we customize the backward elimination procedure [54], [55].

The original backward elimination was developed for supervised learning such as classification, in which we can explicitly define classification error [29]. In such case, we start classification with the full set of features and measure the error. Then, we repeatedly drop the feature that decreases the error most significantly. In contrast, we cannot define the error in the current problem, which is an instance of unsupervised clustering. Thus, we need to modify the original backward elimination.

The procedure we use is as follows. The procedure is iterative, and in each iteration, we perform clustering by dropping a feature at a time. Then we see if there is a feature that causes a change in the silhouette less than δ, a user-specified threshold. If so, this means that this feature does not contribute to the clustering, and then we drop it. Otherwise, we identify the feature that decreases the silhouette by the largest amount and remove it, since it hurts the clustering procedure by lowering the silhouette. (This corresponds to removing the feature that increases the classification error by the largest amount in the supervised learning case.) We do not remove a feature that does not decrease the silhouette.

During backward elimination, we need to fix the clustering algorithm used. In this paper, we use the k-means clustering algorithm [54]. It runs fast and produces reasonable results in many cases, although the k-means algorithm sometimes finds local optima only, as an example of the expectation-maximization algorithm [55].

Fig. 3 shows how the median silhouette value varies during the backward elimination process. For the sake of computational efficiency, we randomly sample 2,000 IO traces and perform the backward elimination process on the selected IO traces. We set the threshold $\delta = 0.2$. The input parameter of this algorithm is k, the number of clusters. For each value of k between 2 and 20, we perform the backward elimination procedure and measure the median silhouette. Overall, the highest peak is observed when $k = 4$ and there remain three features. This means that clustering the trace data with respect to these three features gives the highest silhouette. As shown in Table 1, these three features are F7 (the number of non-random segments), F9 (the ratio of the number of CPs to that of BPs), and F16 (the standard deviation of the starting LBAs of IO requests).

TABLE 2
List of Clustering Algorithms Used

ID	Name	Abbreviation in Fig. 4
A1	k-means	KM
A2	k-medoids	KMD
A3	Fuzzy c-means	FCM
A4	Gustafson-Kessel	GK
A5	Gath-Geva	GG
A6	DBSCAN	DBS
A7	Single-linkage hierarchical	HS
A8	Complete-linkage hierarchical	HC
A9	Average-linkage hierarchical	HA
A10	Ward-linkage hierarchical	HW

5.3 Finding the Best Clustering Algorithm

Based on the three features determined in the previous step, we cluster IO traces by ten different clustering algorithms listed in Table 2. The selection covers most of the existing categories of clustering algorithms [29], [53], [54]: partitioning (A1-A3), fuzzy (A3-A5), density-based (A6), and hierarchical (A7-A10).

To adjust the input parameters of these clustering algorithms, we rely on the silhouette metric. For instance, if one of these clustering algorithms requires the number of clusters as the input parameter (e.g., the k parameter of the k-means algorithm), we try multiple numbers and find the one that yields the highest median silhouette value. For the hierarchical clustering algorithms that report a dendrogram (a hierarchical representation of samples) instead of flat clusters, we use the common transformation technique that cuts the dendrogram at a certain height and group samples in each of the resulting sub-dendrograms into a cluster [54]. We find the right cutting height by maximizing the median silhouette. For other clustering parameters (e.g., the ϵ parameter of DBSCAN that sets the radius of the neighborhood that defines a "dense" region), we also try different values of ϵ and use the one that gives the highest median silhouette value.

Fig. 4 shows the median silhouette values each of the clustering methods used reports. Refer to Table 2 for the definition of the algorithm labels shown in the plot. For clustering, each algorithm uses only the three features (i.e., F7, F9 and F16) determined in the previous backward elimination step. For each clustering method, we try multiple combinations of parameters as stated above, although Fig. 4 shows only some of the results we obtained (including the best ones) due to limited space.

According to Fig. 4, the partition-based algorithms (i.e., A1, A2 and A3) give the best results. The performance of the fuzzy algorithms and that of most of the hierarchical algorithms follow next. The DBSCAN and single-linkage hierarchical

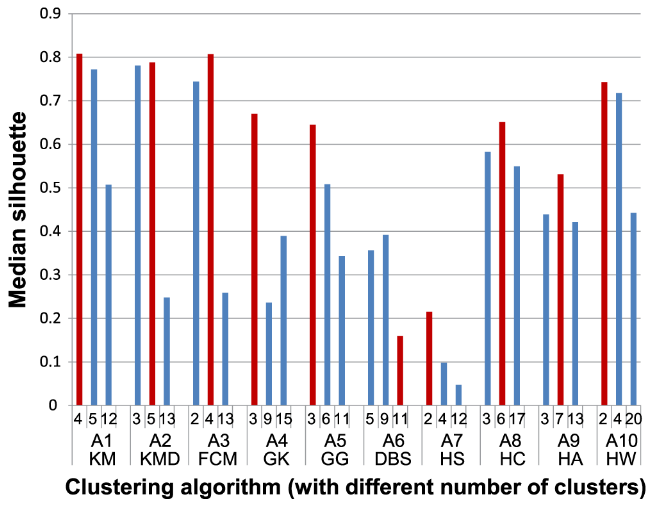

Fig. 4. Comparison of different clustering algorithms (see Table 2 for the definitions of abbreviations).

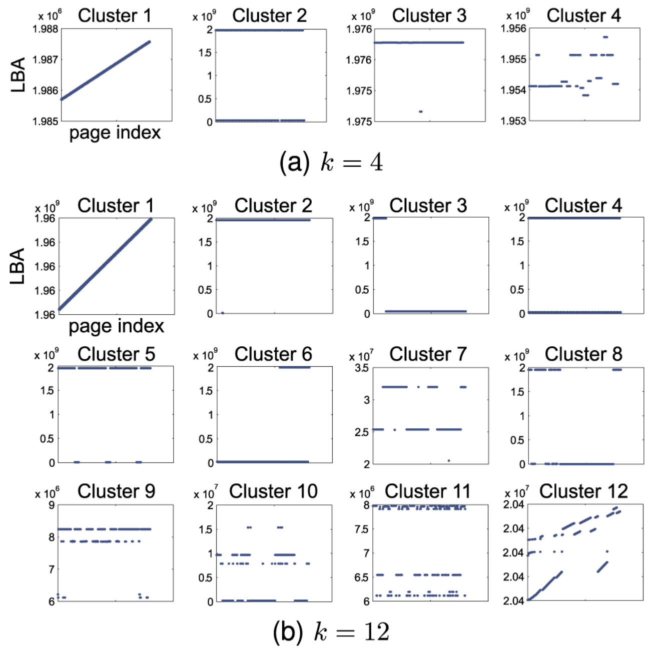

Fig. 5. Samples of preliminary IO trace patterns found by unsupervised clustering (features: F7, F9, F16; algorithm: k-means). (a) $k = 4$ (b) $k = 12$.

algorithms show relatively low performance. In general, DBSCAN can find more complex shapes of clusters than partition-based algorithms can [53]. Nonetheless, the partitioning algorithms produce better results in our experiments, suggesting that IO traces tend to form clusters of rather simple shapes. The single-linkage hierarchical algorithm is known to suffer from the problem of chaining [54] or the formation of long, thin shape of clusters, which are often mere artifacts and fail to uncover the actual internal structure of samples. We may attribute the lowest performance of the single-linkage algorithm in our experiment to this problem. Based on this result, we select the k-means algorithm for downstream analysis due to its performance in terms of the silhouette and its simplicity in implementation.

5.4 Preliminary IO Trace Patterns

Fig. 5 shows two sets of clusters found by our approach from the 58,758 trace samples described in Section 4.2. Note that each cluster represents a "preliminary IO trace pattern," and such patterns will become the grounds for constructing a tree-based model in Section 6. To generate the result shown in the figure, we use the k-means clustering algorithm and the three features determined by the backward elimination procedure. Due to limited space, we only show clusters for write operations; we also found similar cluster formation for read operations.

Fig. 5a shows the representative clusters of write traces found from the clustering with $k = 4$. Recall that the median silhouette is the highest when $k = 4$, as shown in Fig. 4. Among the known IO patterns (i.e., random, sequential and strided), Clusters 1 and 4 in Fig. 5a represent the sequential and random IO patterns, respectively. Cluster 4 is a mixture of segments of short and moderate lengths. The IO requests in Cluster 2 alternate between two LBA regions. In addition, Cluster 2 seems to be related to the strided IO pattern. In Cluster 3, a narrow range of LBAs are accessed except for a short period, during which distant addresses are accessed. Cluster 3 has both short and strided segments (in a magnified view, the longer segment consists of two alternating segments).

Fig. 5b shows the clusters of write traces we found for $k = 12$. Our intention of using a larger value of k is to examine more types of clusters by increasing k. Although the four clusters in Fig. 5a seem to represent some of the known IO patterns well, there may be missing patterns for such a low value of k. According to our experiments, the median silhouette for $k = 12$ is the highest for $k > 6$ when the k-means algorithm is used. For other types of clustering algorithms, we observe a slightly higher median silhouette for A8 with 17 clusters. However, the variation of the number of cluster members is rather high for this A8 case, meaning that some clusters have a large number of samples whereas others have only a few. Thus, we decide to investigate the k-means case for $k = 12$. As expected, the clusters in Fig. 5b show more diverse patterns. Cluster 1 corresponds to sequential IO patterns just as Cluster 1 in the 4-cluster case does. Clusters 2 and 3 consist of a long sequential pattern and a short random pattern (similar to Cluster 3 in the 4-cluster case). Cluster 4 looks similar to Cluster 2 in Fig. 5. Clusters 9-12 represent random patterns, resembling Cluster 4 in the 4-cluster case. Other clusters seem to be related to strided patterns, similarly to combinations of Clusters 3 and 4 in the 4-cluster case.

6 Building a Model for Characterizing IO Workloads

6.1 Defining Classes of IO Traces

Based on the preliminary patterns of IO traces found by clustering, we finalize a set of IO trace patterns. The preliminary clusters give us an idea of what types of patterns arise in IO traces. We perform clustering without reflecting any domain knowledge, in order not to prevent unexpected novel patterns from appearing. In the second phase of our approach, we further streamline the clusters found in the previous step and define a set of patterns. We can observe the following from our clustering result summarized in Fig. 5:

O1: Segments play a role in shaping patterns. Depending on the number of segments and their lengths, we can separate patterns.

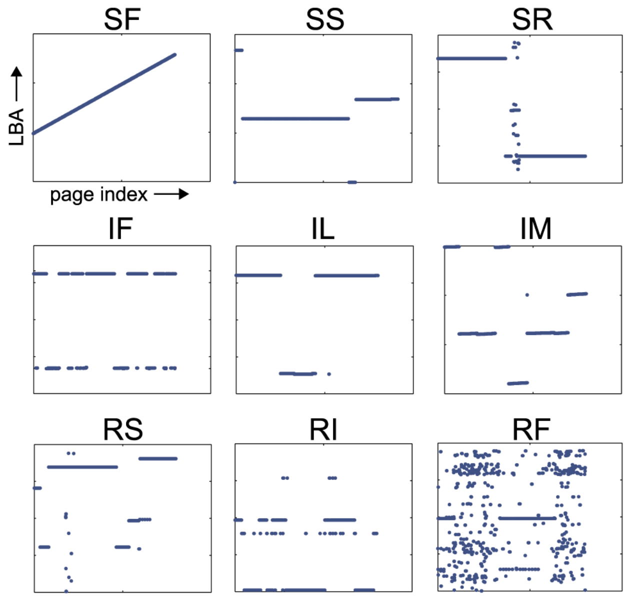

Fig. 6. Sample IO traces for each of the nine classes.

O2: There exist largely three categories of patterns corresponding to sequential, strided and random IO. The sequential patterns include a small number of long segments, whereas the random patterns consist of many short segments. The strided patterns show somewhat regular structures of moderate segments and gaps between them.

O3: Each of the three basic patterns above has a few variants, giving 4 types of patterns in Fig. 5a and 12 types in Fig. 5b.

O4: One such variant is the combined pattern that is a mixture of two or more basic patterns (e.g., a mixture of sequential and strided).

O5: Another variant is the pattern with a basic pattern corrupted by noise-looking short random accesses.

Based on the above observations, we streamline the clusters in Fig. 5 and define 9 classes of IO traces. They are depicted in Fig. 6 with labels and descriptions listed in Table 3.

Although most traces can be represented by (a combination of) the three basic patterns (e.g. sequential, random, and strided), we may encounter a trace that cannot be classified into one of the predefined classes. Should such a trace arise, we can consider it as an 'unknown' class and skip the optimized processing designed for the known classes. Later if the frequency of observing unclassifiable traces increases, we can rerun the characterization and analysis flow to update the predefined classes. Note that the proposed methodology is general and not limited to the nine classes currently presented.

6.2 Representing Classes by Tree

The classification of the 9 patterns shown in Fig. 6 is by design, hierarchical. That is, in the first level of the hierarchy, we can classify IO traces into one of sequential, interleaved and random classes. In the second level, each of these three classes is further divided into three sub-classes, depending on more detailed characteristics. Given that a tree can naturally represent a hierarchical structure, we build a tree-based model to represent the patterns of IO traces. Using a tree-based model gives us several key advantages over more complicated alternatives (e.g., artificial neural networks and the support-vector machine [53], [55]). We can use the constructed tree-based model immediately as a pattern classifier used in the firmware of a storage controller. Implementing a tree-based classifier is straightforward, and its running time overhead is typically negligible, since the classification using a tree model consists of only a small number of comparison operations and does not entail any complex arithmetic operations.

TABLE 3
Classes of IO Traces

ID	Name	Description
SF	Sequential Full	Fully sequential pattern
SS	Sequential Short	Mostly sequential pattern, but with a few short segments
SR	Sequential Random	Sequential pattern mixed with short random patterns
IF	Interleaved† Full	Pattern that consists of a few strided segments
IL	Interleaved Long	Mostly interleaved pattern, but with a few long segments
IM	Interleaved Multilane	More strided segments than IF, with short random patterns
RS	Random Sequential	Mostly random, but with some long sequential segments
RI	Random Interleaved	Mostly random, but with some strided segments
RF	Random Full	Random pattern

† We use the term 'interleaved' in lieu of 'strided' for the sake of avoiding naming conflicts (using 'strided' would make the acronym of 'strided full' (SF) identical to 'sequential full').

Due to the stringent resource requirements of a consumer product (e.g., cost, power consumption, and die size), it is not practically feasible to use high-end microprocessors for SSD controllers. A typical SSD controller uses an embedded processor with a few hundred MHz of clock cycles without a floating-point operation unit. The computational complexity of the classification algorithm used is of critical concern in the SSD controller design and implementation. Employing a more sophisticated classifier is likely to incur additional implementation complexity and runtime penalty with only marginal improvements.

To construct a tree-based model, we need to determine, for each level of the tree, the criterion by which we make a decision. For example, at the root level, we can group fully sequential patterns (SF) into the left node (excluding them from further consideration) and keep on classifying the remaining patterns by going down to the right node. As the decision criterion at the root, a feature that can separate the SF pattern and the rest would be useful in this example.

The set of three features found by backward elimination (i.e., F7, F9 and F16) was successful in discovering clusters in the previous step of our approach, but we see some issues for using this set of features as it is in the tree-based model construction. First, as shown in Fig. 7, the distributions of these three features are all skewed (the i-th diagonal plot shows the histogram of the i-th feature; more explanation of this figure will follow shortly). This property might have been useful in the clustering step, and probably this is why the set of these features was selected as the best combination of features in backward elimination. However, using only those features that have high skewness may produce severely

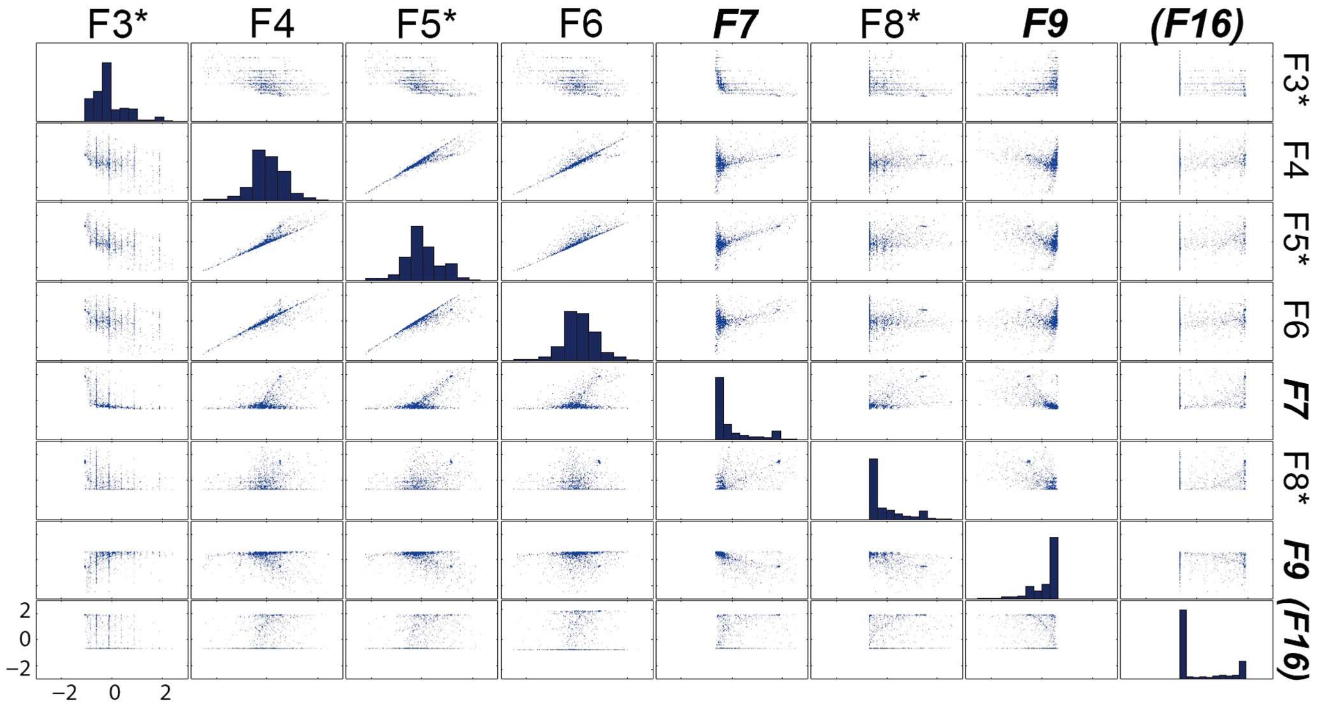

Fig. 7. Scatter-plot matrix [29], [56] of features (F7, F9, F16: selected by the backward elimination; F3, F5, F8: newly added during the tree-based modeling; F16: not considered in the tree-based modeling).

unbalanced classification results. Some classes may have too many samples, whereas others may have only a few. Second, given that the tree-based model is to be implemented as part of the firmware of an IO system eventually, we should not ignore the computational overhead of obtaining features. In this regard, the F16 feature (the standard deviation of the starting LBAs of IO requests) is not ideal due to the need for calculating the standard deviation.

To address these issues, we augment the set of features selected from the clustering procedure as follows: First, we add F3 and F5 into the feature set in order to alleviate the skewness issue. Fig. 7 shows the scatter-plot matrix [29], [56] of some of the 20 features defined in Table 1. In a scatter-plot matrix, the plot drawn at location $(i, j), i \neq j$ is the scatter plot showing the correlation between features i and j, and the i-th diagonal plot is the histogram of feature i presenting its distribution. In our data, F3-F6 are the top four features whose distributions resemble the normal distribution most closely. Among these, we drop F4 and F6 because their pairwise correlations with F5 are high (0.916 and 0.908, respectively), and F5 can be used more conveniently to define the 9 classes listed in Table 3 than these two. Second, instead of F16, we use F8. We can justify this substitution on the following grounds. Most of all, F8 is the last feature removed but F7, F9 and F16. That is, if we had chosen four features by backward elimination, not three, then the feature set would have had F7, F8, F9 and F16. In addition, the two features that have the highest correlation with F16 turn out to be F7 and F8 (0.616 and 0.565, respectively); however, F7 is already in the feature set. Conclusively, replacing F16 by F8 should be a reasonable choice.

Based on this augmented set of features (F3, F5, F7, F8 and F9), we can build a tree-based model for the 9 classes defined in Table 3. First, we can separate the SF class by associating the decision regarding F5 (the number of BPs) with the root node. Then, we can find the SS class by applying F9 (the ratio of the number of CPs to that of BPs) at the second-level node. We can classify the other patterns by using the remaining features downward the tree, as shown in Fig. 8.

Using the decision tree model shown in Fig. 8, we can obtain the information on the quantitative appearance of each of the nine classes of IO traces listed in Table 3. For example, the SF patterns have F5 (the number of break points) values less than 4, whereas the SS patterns have F5 values between 4 and 20, and F9 (the ratio of the number of continued points to that of break points) values less than 0.2.

7 Decision Tree for Classifying IO Traces

Now that we have a tree-based model of IO traces, we can construct a decision-tree-based classifier for classifying IO traces in runtime (Fig. 1). This section explains how to train the classifier and validate it with real traces. Also, we compare the proposed tree-based classifier with other existing state-of-the-art classifiers in terms of classification accuracy and running time. We also consider the impact of the length of trace samples on classification performance.

7.1 Training and Validating the Classifier

We collect labeled examples for each of the 9 classes from the 58,758 trace samples. Following the standard procedure of training and validating a classifier [53], we divide the class-labeled samples into training and validation sets, train the tree-based classifier using the training set and calculate the classification error by the validation set. We perform this step via the 10-fold cross-validation [55]. In this procedure, we divide labeled examples into 10 chunks of equal size, and train and validate the classifier 10 times. For each run, 9 of the 10 chunks are used for training and the remaining one is for

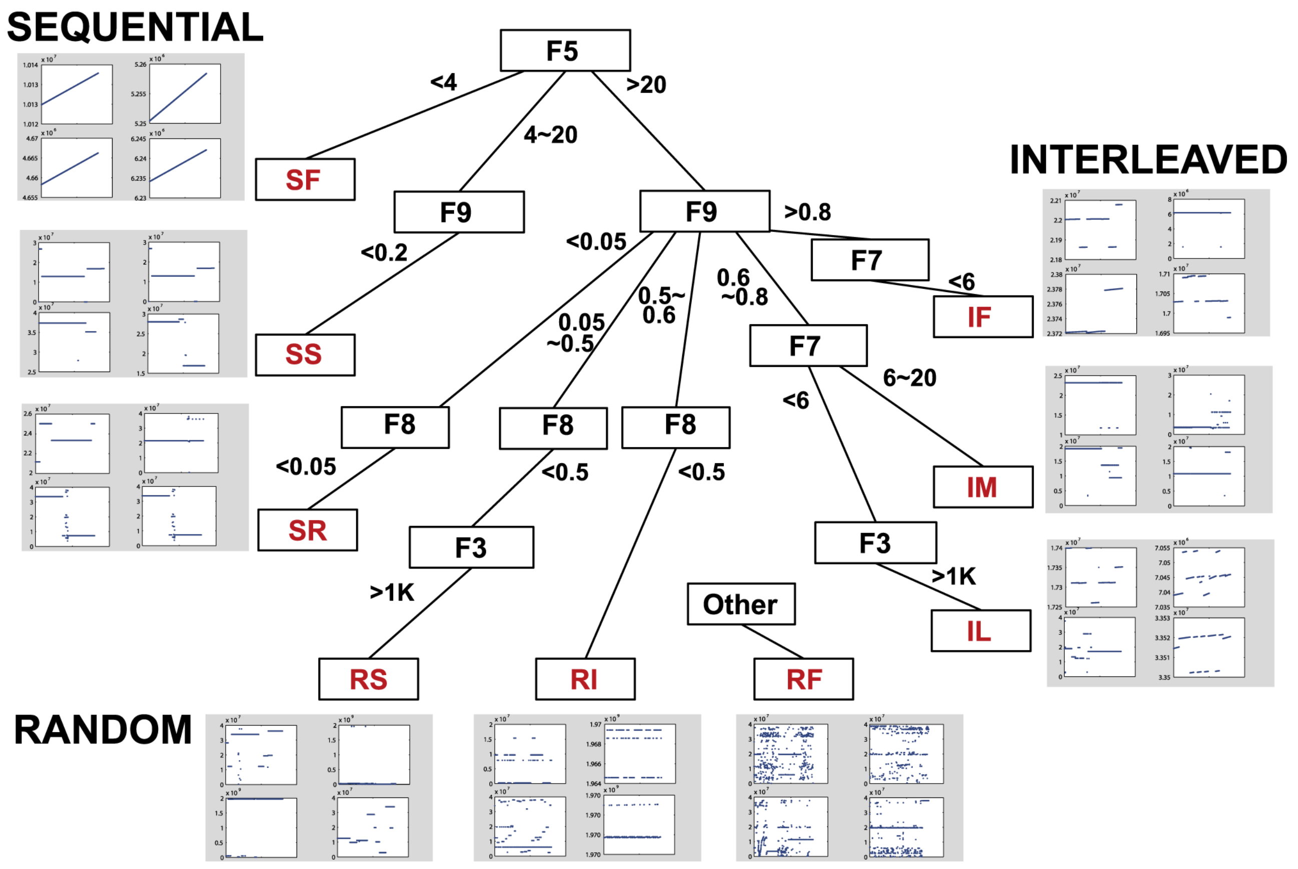

Fig. 8. Proposed decision tree model of IO traces and examples.

validation in turn. We select the parameters of a classifier that give the lowest validation error. Fig. 8 shows the resulting decision tree annotated with the threshold values for making decisions at each node and some example patterns for each class.

7.2 Class Distribution in Real Workloads

To see how the 9 classes defined in Table 3 are distributed in the real IO trace samples we collected, we classify the unlabeled samples not used for training and validation. The set of such samples are often described as the *publication set*. Table 4 lists the breakdown of each of the 9 classes. Approximately half of all the IO trace samples are random patterns, while over 40% of them are interleaved patterns. Among the 9 classes defined in Table 3, the most frequently occurring pattern is the IM class, which consists of multiple strided segments along with some random patterns. The RI class, which has mostly random patterns with some strided patterns, accounts for about 30% of trace samples. The chance of seeing the fully sequential SF pattern is 2.58%, and summing up all the percentages of SF, SS and SR classes yields 4.22%. It would be exciting to see more sequential patterns, given that some of the new storage systems can process sequential data efficiently. It is notable that the strided patterns (the IF, IL and IM classes) accounts for more than 40% altogether. Exploiting such patterns would be beneficial in solid-state drives (SSDs) that can assign each short segment to different channels simultaneously, which can give a steep performance boost. The fraction of fully random patterns (the RF class) is somewhat lower than expected.

7.3 Performance Comparison

Fig. 9 compares the performance of different classifiers in terms of their classification error and running time. The alternative classifiers tested include the support vector machine (SVM) [55], the k-nearest-neighbor classifier [29] and the naïve Bayes classifier [53]. We utilize the WEKA machine learning package [29] for these sets of experiments.

Fig. 9a compares four alternative classification methods in terms of their accuracies relative to that of the tree-based classifier. The accuracy of the logistic-regression-based classifier is the highest, and that of the naïve Bayes classifier is the lowest. The performance of the SVM, one of the most popular classifiers, is similar to that of the best one.

Fig. 9b shows the running time required for classifying 58,758 IO traces by the methods used for comparison.

TABLE 4
Breakdown of 9 Classes in Real Trace Samples (See Table 3 for Definitions of Class ID)

Pattern	Breakdown (%)	Class ID	Breakdown (%)
Sequential	4.22	SF	2.58
		SS	1.22
		SR	0.42
Interleaved	43.4	IF	3.54
		IL	0.21
		IM	39.6
Random	52.4	RS	15.4
		RI	31.3
		RF	5.7

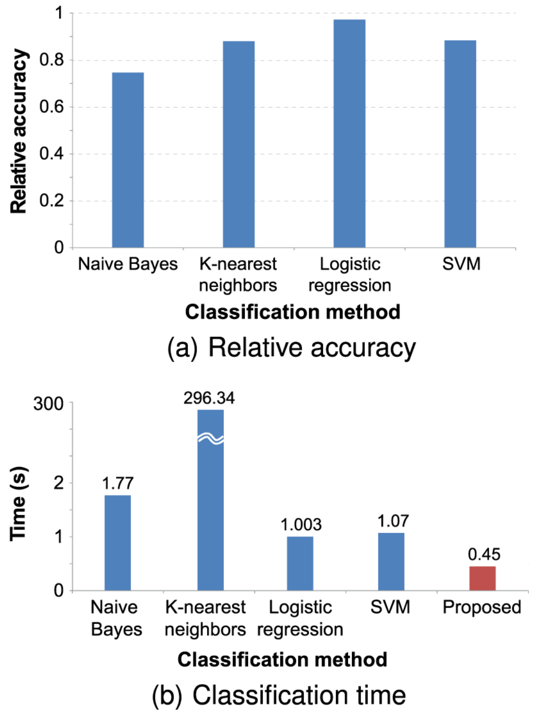

Fig. 9. Performance comparison of classifiers. (a) relative accuracy (b) classification time.

As expected, the proposed method requires the least amount of time to complete the classification task. The advantage in running time obviously comes from the simplicity of the tree-based model. When executed in an embedded CPU, the proposed classifier would further widen the performance gap among the other methods in terms of running time. As for the accuracy in the embedded environment, we need further studies in order to assess the effect of using fixed-point operations for the classifiers under comparison. Still, we expect that the proposed approach would get affected the least, given its simplicity, compared with more sophisticated classifiers such as the SVM, which often involves more complicated arithmetic operations.

For the data we used, the training time of the proposed classifier is faster than that of logistic regression but moderately slower than that of the other classifiers. However, in the current context, it is needless to say that reducing classification time is far more important than accelerating training. As a lazy-learning method that defers all computation until classification [53], the k-nearest-neighbor classifier does not need any training but takes the largest running time, as presented in Fig. 9b, and requires storing all training data, which makes it inappropriate in practice. Overall, the proposed approach outperformed the other classifiers in terms of classification accuracy and time as well as the suitability of firmware implementation.

8 Summary and Conclusion

We have described novel IO workload characterization and classification schemes based on the data mining approach. For IO workload clustering, k-means algorithm yields the best silhouette value despite its simplicity. Among the twenty features of the IO workload, we select three features to form a minimal feature set: the number of non-random segments, the ratio of the number of continued points and the number of break points, and the standard deviation of the starting LBAs of IO requests. As IO classes, we establish nine essential workload classes: three random, three sequential and three mixture classes in legacy sense. Finally, we develop a classification model of IO traces and construct a pattern classifier based on the model for classifying IO traces in runtime. We use a tree-based model that naturally fits the hierarchical nature of workloads. Based on the model, we also develop a decision tree for classifying IO trace in the storage stack of the operating systems or the storage device firmware. An SSD controller can effectively exploit this classification model to characterize incoming workloads so that it can adaptively apply it to essential tasks such as garbage collection, wear leveling, and address mapping, tailoring these tasks to each workload class. Later in our future work, we plan to develop FTL techniques that can effectively utilize the classification model proposed in this work.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science, ICT and Future Planning) [No. 2011-0009963 and No. 2012-R1A2A4A01008475] and in part by the IT R&D program of Korea Evaluation Institute of Industrial Technology (KEIT) funded by the Korean government (Ministry of Trade, Industry and Energy) [No. 10035202].

References

[1] "SSD Prices Continue to Plunge," http://www.computerworld.com/s/article/9234744/SSD_prices_continue_to_plunge, 2012.
[2] "SSD Prices Are Low—and They'll Get Lower," http://arstechnica.com/gadgets/2012/12/ssd-prices-are-low-and-theyll-get-lower/, 2012.
[3] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," *Proc. ACM Special Interest Group on Management of Data (SIGMOD) Int'l Conf. Management of Data*, pp. 1075-1086, 2008.
[4] S.R. Hetzler, "The Storage Chasm: Implications for the Future of HDD and Solid State Storage," *Proc. IDEMA Symp. What's in Store for Storage, the Future of Non-Volatile Technologies*, 2008.
[5] S.-W. Lee, B. Moon, and C. Park, "Advances in Flash Memory SSD Technology for Enterprise Database Applications," *Proc. 35th ACM Special Interest Group on Management of Data (SIGMOD) Int'l Conf. Management of Data*, pp. 863-870, 2009.
[6] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," *ACM Trans. Embedded Computing Systems*, vol. 6, pp. 18-46, July 2007.
[7] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *Special Interest Group on Operating Systems (SIGOPS) Operating Systems Rev.*, vol. 42, pp. 36-42, Oct. 2008.
[8] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compactflash Systems," *IEEE Trans. Consumer Electronics*, vol. 48, no. 2, pp. 366-375, May 2002.
[9] H. Kwon, E. Kim, J. Choi, D. Lee, and S.H. Noh, "Janus-FTL: Finding the Optimal Point on the Spectrum between Page and Block Mapping Schemes," *Proc. 10th ACM Int'l Conf. Embedded Software*, pp. 169-178, 2010.
[10] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, pp. 229-240, 2009.

[11] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance Enhancement of Flash-Memory Storage, Systems: An Efficient Static Wear Leveling Design," *Proc. 44th ACM/IEEE Design Automation Conf. (DAC'07)*, pp. 212-217, 2007.

[12] L.-P. Chang, "On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems," *Proc. ACM Symp. Applied Computing*, pp. 1126-1130, 2007.

[13] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf.*, pp. 57-70, 2008.

[14] J.-W. Hsieh, L.-P. Chang, and T.-W. Kuo, "Efficient On-Line Identification of Hot Data for Flash-Memory Management," *Proc. ACM Symp. Applied Computing*, pp. 838-842, 2005.

[15] L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 3-14, 1998.

[16] J. Zedlewski et al., "Modeling Hard-Disk Power Consumption," *Proc. 2nd USENIX Conf. File and Storage Technologies*, vol. 28, pp. 32-72, 2003.

[17] A. Riska and E. Riedel, "Disk Drive Level Workload Characterization," *Proc. USENIX Ann. Technical Conf.*, pp. 97-103, 2006.

[18] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of Storage Workload Traces from Production Windows Servers," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC)*, pp. 119-128, 2008.

[19] D. Molaro, H. Payer, and D. Le Moal, "Tempo: Disk Drive Power Consumption Characterization and Modeling," *Proc. IEEE 13th Int'l Symp. Consumer Electronics (ISCE'09)*, pp. 246-250, 2009.

[20] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi, "A High Performance Controller for NAND Flash-Based Solid State Disk (NSSD)," *Proc. 21st IEEE Non-Volatile Semiconductor Memory Workshop (NVSMW)*, pp. 17-20, 2006.

[21] Y. Lee, L. Barolli, and S.-H. Lim, "Mapping Granularity and Performance Tradeoffs for Solid State Drive," *J. Supercomputing*, vol. 65, pp. 1-17, 2012.

[22] G. Wu and X. He, "Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality," *Proc. 7th ACM European Conf. Computer Systems*, pp. 253-266, 2012.

[23] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H.-U. Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in SSDs: Model and Quantitative Analysis," *Proc. IEEE 28th Symp. Mass Storage Systems and Technologies (MSST)*, pp. 1-12, 2012.

[24] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 91-103, 2011.

[25] Q. Yang and J. Ren, "I-CASH: Intelligently Coupled Array of SSD and HDD," *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 278-289, 2011.

[26] J. Ren and Q. Yang, "A New Buffer Cache Design Exploiting Both Temporal and Content Localities," *Proc. IEEE 30th Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 273-282, 2010.

[27] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *Proc. ACM SIGOPS Operating Systems Rev.*, vol. 34, no. 5, 2000, pp. 150-159.

[28] A. Lakhina, M. Crovella, and C. Diot, "Mining Anomalies Using Traffic Feature Distributions," *Proc. ACM Special Interest Group on Data Communication (SIGCOMM)*, pp. 217-228, Aug. 2005.

[29] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[30] Y. Won, H. Chang, J. Ryu, Y. Kim, and J. Ship, "Intelligent Storage: Cross-Layer Optimization for Soft Real-Time Workload," *ACM Trans. Storage*, vol. 5, no. 4, pp. 255-282, 2006.

[31] N. Park, W. Xiao, K. Choi, and D.J. Lilja, "A Statistical Evaluation of the Impact of Parameter Selection on Storage System Benchmarks," *Proc. 7th IEEE Int'l Workshop on Storage Network Architecture and Parallel I/O (SNAPI)*, 2011.

[32] M.G. Baker, J.H. Hartman, M.D. Kupfer, K. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," *Proc. 13th ACM Symp. Operating System Principles*, pp. 198-212, 1991.

[33] S. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller, "Self-Similarity in File Systems," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 141-150, 1998.

[34] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the Unix 4.2 bsd File System," *Proc. 10th ACM Symp. Operating Systems Principles (SOSP)*, pp. 15-24, 1985.

[35] E. Riedel, M. Kallahalla, and R. Swaminathan, "A Framework for Evaluating Storage System Security," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, 2002.

[36] M.F. Arlitt and C.L. Williamson, "Web Server Workload Characterization: The Search for Invariants (Extended Version)," *Proc. ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, pp. 126-137, 1996.

[37] K. Krishna and Y. Won, "Server Capacity Planning under Web Traffic Workload," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 5, pp. 731-747, Sept./Oct. 1999.

[38] B. Knighten, "Detailed Characterization of a Quad Pentium Pro Server Running tpc-d," *Proc. IEEE Int'l Conf. Computer Design*, pp. 108-115, 1999.

[39] T. Harter, C. Dragga, M. Vaughn, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications," *Proc. Symp. Operating Systems Principles (SOSP)*, pp. 71-83, 2011.

[40] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 173-186, 2004.

[41] M. Sivathanu, V. Prabhakaran, F.I. Popovici, T.E. Denehy, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, vol. 3, pp. 73-88, 2003.

[42] "Iometer," http://www.iometer.org/, last accessed on Feb. 26, 2014.

[43] N. Agrawal, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Generating Realistic Impressions for File-System Benchmarking," *Proc. 7th Conf. File and Storage Technologies*, pp. 125-138, 2009.

[44] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan, "Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking," *Proc. 3rd USENIX Conf. File and Storage Technologies*, pp. 45-58, 2004.

[45] "Filebench," http://sourceforge.net/projects/filebench/.

[46] D. Anderson, "Fstress: A Flexible Network File Service Benchmark," Duke Univ., Technical Report TR-2001-2002, 2002.

[47] L. Bouganim, B.T. Jonsson, and P. Bonnet, "uFLIP: Understanding Flash IO Patterns," *Proc. Classless Inter-Domain Routing (CIDR)*, 2009.

[48] M. Bjorling, L. De Folgoc, A. Mseddi, P. Bonnet, L. Bouganim, and B. Jonsson, "Performing Sound Flash Device Measurements: Some Lessons from uFLIP," *Proc. ACM Special Interest Group on Management of Data (SIGMOD) Int'l Conf. Management of Data*, pp. 1219-1222, 2010.

[49] G. Goodson and R. Iyer, "Design Tradeoffs in a Flash Translation Layer," *Proc. HPCA Workshop on the Use of Emerging Storage and Memory Technologies*, Jan. 2010.

[50] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *ACM Trans. Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[51] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices," *IEEE Trans. Computers*, vol. 58, no. 6, pp. 744-758, June 2009.

[52] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. 6th USENIX Conf. File and Storage Technologies*, pp. 1-14, 2008.

[53] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Pub, 2011.

[54] P.N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Pearson Addison Wesley, 2006.

[55] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2004.

[56] S. Few, *Show Me the Numbers: Designing Tables and Graphs to Enlighten*, 2nd ed. Analytics Press, 2012.

Bumjoon Seo is an advisory researcher with the Emerging Technology Laboratory at Samsung SDS, Co., LTD., Seoul, Korea. His research interests include applications of data mining and machine learning to storage systems design.

Sooyong Kang received the BS degree in mathematics and the MS and PhD degrees in computer science, from Seoul National University (SNU) in 1996, 1998, and 2002, respectively. He was then a postdoctoral researcher in the School of Computer Science and Engineering, SNU. He is now with the Department of Computer Science and Engineering, Hanyang University, Seoul. His research interests include operating systems, multimedia systems, storage systems, flash memories and next generation nonvolatile memories, and distributed computing systems.

Jongmoo Choi received the BS degree in oceanography from Seoul National University, Korea, in 1993 and the MS and PhD degrees in computer engineering from Seoul National University in 1995 and 2001, respectively. He is currently an associate professor with the Department of Software, Dankook University, Korea. Previously, he was a senior engineer at Ubiquix Company, Seoul, Korea, from 2001 to 2003. He held a visiting faculty position at the University of California, Santa Cruz, from 2005 to 2006. His research interests include operating system, file system, mobile storage, and virtualization.

Jaehyuk Cha is a professor with the Department of Computer Science and Engineering at Hanyang University, Seoul, Korea.

Youjip Won received the BS and MS degrees in computer science from the Seoul National University, Korea, in 1990 and 1992, respectively. He received the PhD degree in computer science from the University of Minnesota, Minneapolis, in 1997. After receiving the PhD degree, he joined Intel as a server performance analyst. Since 1999, he has been with the Department of Computer Science and Engineering, Hanyang University, Seoul, Korea, as a professor. His research interests include operating systems, file and storage subsystems, multimedia networking, and network traffic modeling and analysis.

Sungroh Yoon (S'99-M'06-SM'11) received the BS degree in electrical engineering from Seoul National University, Korea, in 1996, and the MS and PhD degrees in electrical engineering from Stanford University, California, in 2002 and 2006, respectively. From 2006 to 2007, he was with Intel Corporation, Santa Clara, California. Previously, he held research positions with Stanford University, and Synopsys, Inc., Mountain View, California. He was an assistant professor with the School of Electrical Engineering, Korea University, Seoul, South Korea, from 2007 to 2012. Currently, he is an assistant professor with the Department of Electrical and Computer Engineering, Seoul National University. He is the recipient of the 2013 IEEE/IEEK Joint Award for Young Engineers. His research interests include emerging computing and storage systems and high-performance bioinformatics. He is a senior member of the IEEE.